7-2019

Exploration of GPU Cache Architectures Targeting Machine Learning Applications

Gerald Kotas
gk3952@rit.edu

Exploration of GPU Cache Architectures Targeting Machine Learning Applications

GERALD KOTAS

Exploration of GPU Cache Architectures Targeting Machine Learning Applications

GERALD KOTAS

July 2019

A Thesis Submitted
in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in
Computer Engineering

R·I·T | KATE GLEASON *College of* ENGINEERING

Department of Computer Engineering

Exploration of GPU Cache Architectures Targeting Machine Learning Applications

GERALD KOTAS

Committee Approval:

Dr. Sonia López Alarcón *Advisor* Date
RIT, Department of Computer Engineering

Dr. Marcin Łukowiak Date
RIT, Department of Computer Engineering

Dr. Roy Melton Date
RIT, Department of Computer Engineering

Acknowledgments

First and foremost, I would like to thank my advisor Dr. Sonia López Alarcón for her guidance and support throughout this entire process. This has been a long and arduous process and I am thankful for all of the help along the way. I would also like to thank Dr. Marcin Łukowiak and Dr. Roy Melton for serving on my thesis committee.

Next, I want to thank my friends for continually pushing me and driving me in the right direction. Finally, I would like to thank my family for their support and understanding for me for finishing this thesis.

Abstract

The computation power from graphics processing units (GPUs) has become prevalent in many fields of computer engineering. Massively parallel workloads and large data set capabilities make GPUs an essential asset in tackling today's computationally intensive problems. One field that benefited greatly with the introduction of GPUs is machine learning. Many applications of machine learning use algorithms that show a significant speedup on a GPU compared to other processors due to the massively parallel nature of the problem set. The existing cache architecture, however, may not be ideal for these applications. The goal of this thesis is to determine if a cache architecture for the GPU can be redesigned to better fit the needs of this increasingly popular field of computer engineering.

This work uses a cycle accurate GPU simulator, Multi2Sim, to analyze NVIDIA GPU architectures. The architectures are based on the Kepler series, but the flexibility of the simulator allows for emulation of newer features. Changes have been made to source code to expand on the metrics recorded to further the understanding of the cache architecture. Two suites of benchmarks were used: one for general purpose algorithms and another for machine learning. Running the benchmarks with various cache configurations led to insight into the effects the cache architecture had on each of them. Analysis of the results shows that the cache architecture, while beneficial to the general purpose algorithms, does not need to be as complex for machine learning algorithms. A large contributor to the complexity is the cache coherence protocol used by GPUs. Due to the high spacial locality associated with machine learning problems, the overhead needed by implementing the coherence protocol has little benefit, and simplifying the architecture can lead to smaller, cheaper, and more efficient designs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphics processing units (GPUs) may have started out as dedicated processors to handle the graphics of a computer, but they have evolved to a much more critical role. Computing and displaying intricate graphics is major application for GPUs; however, general purpose computing has become an increasingly popular application ever since NVIDIA introduced CUDA, its general purpose application programming interface (API) [3]. These general purpose GPUs, known as GPGPUs, use the CUDA development environment to integrate with the central processing unit (CPU) of a computer. This integration of two different types of processors is the basis of heterogeneous computing.

Heterogeneous computing is the idea that multiple different processor types work together as a single system. This is powerful because it leverages the fact the different processors excel at their own types of target applications. When multiple different processors are available, the best processor for the job is selected, and the best performance can be achieved for many different problems. GPUs are a key component to almost all heterogeneous systems due to their unique ability to handle massively parallel problems in a very efficient manner. Today's problems all deal with massive amounts of data that need to be processed, making the GPU a viable option for many of the cutting edge applications. This is particularly applicable to machine learning.

1.1 Motivation

A large and ever-growing field in computer engineering is machine learning. The nature of machine learning applications is large data and parallel execution. These characteristics point to GPUs being a practical option when choosing which processor to use. The motivation of this work is to focus on the cache performance of the GPU when executing machine learning algorithms. Since GPUs are designed to handle a wide variety of problem types, the specific use case of machine learning may not be getting all of the performance speedup that could be possible, or the architecture has unneeded resources that add to the production or running cost.

The GPU cache architecture was modeled after that of the CPU, with slight modifications. The work of this research involves running many general purpose and machine learning benchmarks on various different cache configurations. By analyzing the results, an alternative cache architecture may be found that suits machine learning applications better than the general purpose one that already exists in GPUs.

This research gets its foundation from the work of Nimkar in [4]. He created a suite of machine learning benchmarks to simulate the workload of actual machine learning applications. Since the GPU simulator cannot run full machine learning programs, a set of common algorithms used by a wide variety of machine learning applications were compiled. This suite was used in his work which found that there was different cache behavior when compared to the general purpose benchmark suite. Nimkar found that the L2 cache hit ratio is always low, around 50%. He also found that the most recently used cache line accounts for a majority of the hits leading to the conclusion that a large L2 cache isn't necessary for machine learning applications. The work of this thesis dives more into the reason why this is the case and reconfigures the cache architecture to better suit the machine learning benchmarks. An area of focus is the L2 cache due to its large size in manufacturing and its higher power consumption [5].

Chapter 2

Background

2.1 GPU Architectures

The architecture of a GPU differs greatly from that of the conventional CPU. This alternative architecture is what allows for the massively parallel abilities of the GPU. The two main producers of GPUs are NVIDIA and AMD. These two companies use different terminology and frameworks for their GPUs. NVIDIA uses its own framework called CUDA [3] while AMD uses the open source OpenCL. Although these two have strong parallelism, the rest of this research will be referencing only CUDA terminology.

A big difference between a CPU and GPU is that a GPU can support thousands of threads whereas a CPU can handle only tens of threads. These thousands of threads are executed in groups of 32 called warps. Each warp has its own program counter, meaning each thread inside the warp executes the same instruction. There is a bit for each thread to indicate if the thread is enabled for that instruction. Execution of warps is handled by warp managers. A manager picks a warp from a pool of ready warps. The readiness of each warp is monitored by a scoreboard. As long as there is at least one warp in the pool, execution of the warp can happen immediately [6].

Threads are structured into blocks. The shape can be up to three dimensions, and the size in each direction is limited by the CUDA compute capability. The maximum size of a block in the x, y, and z directions, since the 2.0 compute capability through the latest of 7.5, are 1024, 1024, and 64, respectively. The total number of threads per block is also limited

to 1024 [3]. This means that the maximum size in every direction is not simultaneously achievable. The number of dimensions and the size are up to the programmer on a per kernel basis and should map to the problem size for greatest performance. Similarly, blocks are structured in a grid. The grid can also be up to three dimensions. The CUDA compute capability also dictates the size, albeit a much larger limit. The maximum size in the x-direction is $2^{31} - 1$. The y- and z-directions each have a maximum of 65535. Again, the shape of the grid is set per kernel and should match the scope of the problem.

During a kernel's execution, blocks are assigned to streaming multiprocessors (SMs). Each SM is composed of many streaming processors (SPs) that handle the arithmetic execution of the warps. These SPs are also known as CUDA cores, and the number available varies between architectures. Figure 2.1 shows these as the rows of green squares in the GPU layout. There are also resources, such as a register file and caches, that are divided between all of the blocks currently executing on the SM. There are three limiting factors to the number of blocks that can be assigned to a SM at once. These are the register file, shared memory, and the thread limit per SM. Every block requires the same amount of shared memory as well as the same number of registers and threads. As long as there are enough of each available, another block may be assigned to the SM. Blocks allocate the space for as long as their warps need to complete execution of the kernel. After that, the resources are freed and another block can be assigned, if there are any waiting for assignment. Ideally, the utilization of each of these resources would be 100%; however, this is unlikely because the amount that each block requires may not evenly divide the size that is available.

Figure 2.1 easily shows the difference between the two processor architectures. The CPU has a handful of arithmetic logic units (ALUs) and a single control unit to control them. There is a single, large cache that buffers the data before needing to access the dynamic random access memory (DRAM). The GPU architecture is structured in a much different way. Firstly, the number of computational units is greatly increased, as shown

by the large number of green boxes in the figure. Secondly, there are multiple control units, each one controlling a subset of all of the computational units. Each control unit is accompanied by its own local cache. The size of this cache is much less than the CPU's cache.

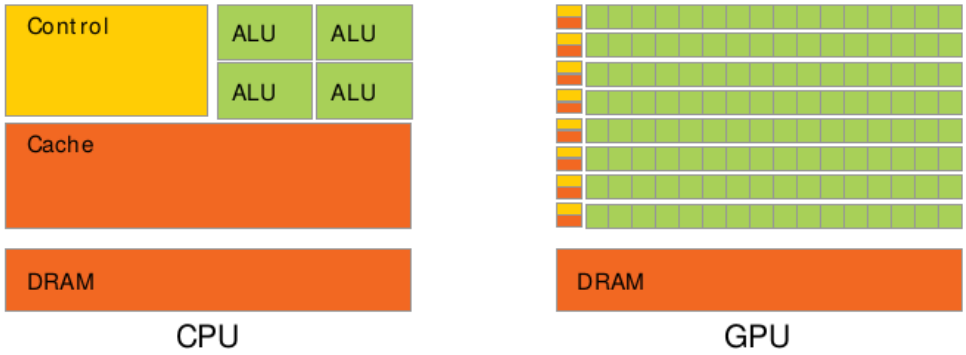


Figure 2.1: CPU vs GPU Architecture [1]

Since the GPU deals with a lot of data, it makes sense that the memory hierarchy is designed for large memory accesses. The cost of accessing global memory is very high. This large latency is masked by the vast number of threads that are executing simultaneously. Once a thread needs to access memory, its access is coalesced with other threads in the same warp, and only one memory access is necessary. The memory bus is able to transfer a half-warp's worth of data in one transaction. Once a warp is waiting for its data, another warp takes its place and begins its execution. The number of warps waiting for execution should mask the latency associated with accessing memory. This is how the GPU achieves its high parallel performance by having enough threads to be continually computing without experiencing memory latency.

The graph in Figure 2.2 illustrates the performance when varying the number of threads. The graph shows three regions labeled MC Region, Valley, and MT Region. The first region is where the CPU operates. The name comes from the many-cores that are on a standard CPU. There is a peak performance limited by the size of the cache and number of threads. Once there are too many threads, the size of the cache can no longer mask the accesses to

memory, and the performance starts to degrade. The region where this decreased performance exists is called the valley. Eventually, the number of threads increases to a point where there are enough threads executing to cover the latency of accessing memory on cache misses. This is known as the many-thread region and is where a GPU operates efficiently. The performance continues to increase until it reaches its peak which is limited by memory bandwidth [7].

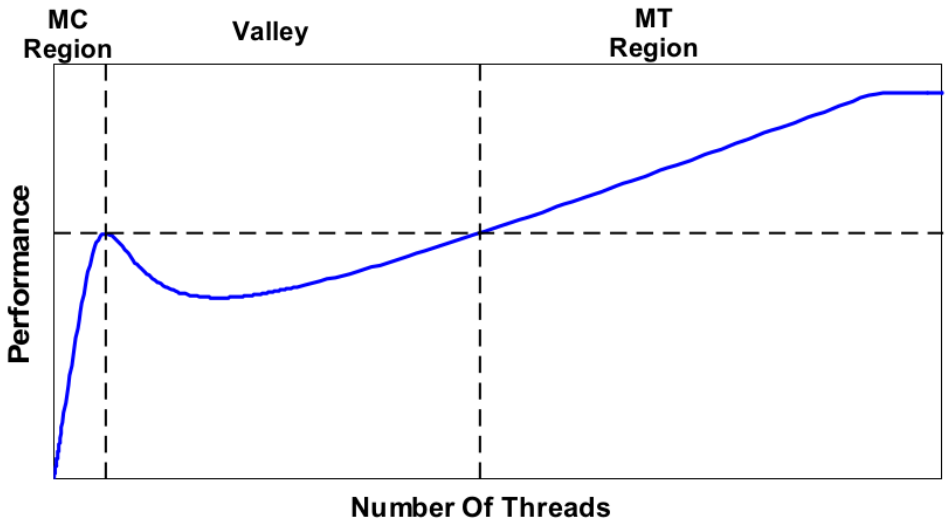


Figure 2.2: Performance Regions of a Unified Many-Core (MC) Many-Thread (MT) Machine based on Number of Threads [7]

To reduce the latency of global memory, the GPU is equipped with multiple caches. Each SM has its own local constant, texture, and L1 data caches. There are L2 data caches shared between all SMs that provide an additional layer of cache before going off to global memory. The constant and texture caches are for the read-only memories. These are read-only for the scope of the kernel. It is up to the CPU to populate these memories for the kernel to use. The L1 and L2 caches are for the data that the GPU uses during its execution. The sizes of these caches are not comparable to that of a CPU since the CPU uses a large cache to cover memory latency while the GPU uses the large number of threads. In NVIDIA architectures Fermi, Kepler, and the newest Volta, the L1 cache shares memory space with shared memory [8, 9]. In the Maxwell and Pascal architectures, these two have

their own dedicated spaces [10, 11] outside of the SM is the L2 cache. Along with this is the page table. The page table maps virtual addresses to physical addresses. The cache for the page table is the translation look-aside buffer (TLB). A miss in the TLB means an access to global memory [10].

2.2 Cache Architecture

2.2.1 Cache Replacement Policies

Since caches have a limited size, it is inevitable that the cache will become full. When a new block of data needs to be placed into cache, an existing block needs to be overwritten. The process that chooses which cache line gets replaced is handled by the cache replacement policy. The ideal replacement policy would be to replace the cache line that is not going to be used for the longest time in the future. Since it is not possible to know when the blocks of data are going to be used in the future, other policies must be used to get close to this perfect scenario. There are several different replacement policies, each with their own benefits and drawbacks.

The simplest policy to implement is random replacement. This does just what the name implies: it replaces a cache line at random. No information needs to be kept on any of the cache lines, which makes it really lightweight. Picking a cache line at random is most closely achieved by picking a number at random using a pseudorandom number generator. The simplicity of this policy makes it a valid possibility for microcontrollers [12]. Another replacement policy is the first in, first out (FIFO) policy. This replacement policy uses the same idea as the queue data structure. The oldest cache line gets replaced with the new data. This policy requires some data to be stored about the cache lines; timestamps are needed to determine which line holds the oldest data. With the slight bit of overhead, this policy performs better when a small subset of data gets reused in quick succession [13].

The least recently used (LRU) replacement policy is the most well known and widely

used replacement policy. This is similar to the FIFO policy, but instead of replacing the oldest added cache line, the cache line with the oldest access time gets replaced. To know the access information for each cache line, more overhead is required. The cache lines are stored in a descending list based on last access time. The last element in the list gets removed and the new block of data gets added to the front of the list. In addition to the replacement updates, an update is needed whenever there is a hit in cache. The cache line that was hit needs to be moved to the front of the list. This policy ensures data that has a high access frequency is always in cache, no matter how long it has been in there [4]. With the popularity of this policy, there are a lot of derivations that improve on some aspect of it, such as simplifying the amount of overhead needed but still retaining as much performance as possible [13].

2.2.2 Cache Write Policies

All bytes of data get written to cache and eventually main memory when a store instruction is processed. Writing to cache always happens as soon as the instruction is processed; however, the time at which it is written to main memory is dictated by a cache write policy. There are two main policies that are implemented in most processing units. The first is the simpler of the two and is called write through. With this policy in place, all writes to cache are synchronized to writes with main memory. This policy ensures that values in cache are always up-to-date with their values in main memory. By having a redundancy, since values are the same in cache and memory at all times, there is less risk for data corruption. Another benefit to this simple write policy is that the cache coherence protocol that is needed is much simpler as well [14]. All that is needed to keep coherency is an invalidation request that is sent to other caches whenever a store is processed. When a cache line gets evicted, there is no additional work needed since the values are already written through to memory.

The second, more widely implemented write policy is write back. This policy uses a

more complex cache coherence protocol to help reduced unneeded writes to memory, since that is a large source of latency. In addition to having a valid bit, tag, and data field, the cache line also needs a dirty bit. This bit signifies that the data field has been written since being read into cache, and it also means that value in memory is out of date. This is because data doesn't get written back to memory until it is evicted from cache. Because the most up-to-date data isn't always in main memory, much more complex coherence protocols need to implemented to handle this. Section 2.2.3 goes into more detail about these protocols. Just like the write through policy, when a store is processed with the write back policy, an invalidation request is sent to other caches. Another case the protocol needs to handle is when a cache requests data, it can't just read it from memory. Rather, a request must be sent to all caches since one of them might have a modified copy. The heart of the write back policy is during eviction of cache lines. If the data isn't dirty, (i.e. the dirty bit is 0), then the data can be overwritten. When the dirty bit is 1, however, the cache line's data must be written back to memory before the new data can overwrite it. The idea of waiting to write the data back to memory is known as lazy writing, and this is where the write back policy saves on its write performance.

Another part of write policies is what to do when a write misses cache, aptly known as the write-miss policy. There are two possible options: no-write allocate and write allocate. The write-miss policies are interchangeable with the cache write policies, but typically write through uses no-write allocate while write back uses write allocate. With no-write allocate, a write that misses cache updates the value only in memory. It is not written into cache. Write allocate will read the data into cache first, then update it and set the dirty bit. This is a little more complicated but improves performance for subsequent reads or writes because the data will be present in cache.

2.2.3 Cache Coherence Protocols

In multiprocessor systems, each processor has its own local cache. These caches can have different values for the same address in main memory depending on when the data was read from memory. For example, if two processors load the same value from memory and both modify it before storing it back the memory, one of the values will be overwritten and the other value is lost. This problem is known as cache incoherence. There are multiple different protocols that attempt to keep coherency between the caches so when one processor modifies a value, the change is propagated through the rest of the system without having to waste time accessing main memory.

A basic cache coherence protocol is MSI named for the three possible states in it: *modified*, *shared*, and *invalid*. An empty cache line starts in the *invalid* state. If a load instruction occurs, the cache line is moved up into the *shared* state. This state allows for reading permissions but not writing permissions. To gain write permissions, for a store instruction for example, the cache line must be in the *modified* state by issuing a write request. The downside to moving into the *modified* state is that all other cache lines with that data must be invalidated. On a similar note, moving into the *shared* state requires all cache lines in the *modified* state to be downgraded into the *shared* state. These three states and the rules of transitioning are enough to ensure a coherent cache.

There are two sources of transition actions, the processor and internal requests. Processor transition actions are load and store instructions. The internal requests are the rules needed to keep the coherency between the caches. Depending on the action, additional internal requests may be issued to guarantee coherence. For example, after the processor action of a store instruction, a write request is issued to invalidate all other copies of that data so only one copy is modified at a time.

To further explain the MSI coherence protocol, Table 2.1 shows the contents of three caches and main memory after processing several instructions. The table is has seven columns. The first is the step number, and the second is the current instruction being

processed. The next three are for the contents of each of the caches. Their values are shown as well as the current state shown in parentheses. The next column shows the contents of main memory. Finally, the last column shows the current request that is on the bus along with the module that initiated in parentheses.

Table 2.1: MSI Cache Coherence Example

Step	Instruction	Cache 0	Cache 1	Cache 2	Memory	Bus
0	–	(I)	(I)	(I)	3	–
1	Cache 0 Read	(I)	(I)	(I)	3	Read Request (C0)
2		3 (S)	(I)	(I)	3	Send Data (Memory)
3	Cache 0 Write	3 (S)	(I)	(I)	3	Write Request (C0)
4		6 (M)	(I)	(I)	3	–
5	Cache 1 Read	6 (M)	(I)	(I)	3	Read Request (C1)
6		6 (S)	6 (S)	(I)	3	Send Data (C0)
7		6 (S)	6 (S)	(I)	6	Write Back (C0)
8	Cache 2 Read	6 (S)	6 (S)	(I)	6	Read Request (C2)
9		6 (S)	6 (S)	6 (S)	6	Send Data (C1)
10	Cache 1 Write	6 (S)	6 (S)	6 (S)	6	Write Request (C1)
11		(I)	9 (M)	(I)	6	–

Step 0 shows the initial configuration of the memory; each cache is in the *invalid* state, and the value in memory is 3. The first instruction is a read issued to cache 0. This takes two steps to complete. The first is a read request sent by cache 0 to the other caches requesting for a copy of the data. Since the other caches are in the *invalid* state, this request gets ignored. Cache 0 gets a copy of the data from main memory in the *shared* state.

The next instruction is a write to cache 0 because the processor modified the data. Since the copy in cache only has read permissions, a write request needs to be sent to the other caches to invalidate their copies, if they have them. The other caches are already in the *invalid* state, so the request is ignored. Once that is completed, cache 0 is free transition into the *modified* state as shown in step 4 when the value gets changed to 6.

The third instruction is a read issued for cache 1. To get a copy of the data, cache 1 issues a read request to the other caches before having to get it from memory. This time the read request is acknowledged by cache 0. Cache 0 responds by sharing a copy of the data

which downgrades the copy in cache 0 to the *shared* state since it no longer can have write permissions. By downgrading states, cache 0 also needs to write the data back to main memory so it also has the updated copy. These three steps are shown as steps 5-7, and the final result is caches 0 and 1 both have the updated copy in the *shared* state.

The next instruction is for cache 2, and it is also a read. The first step is to issue a read request to other caches. Since both of the other caches have a copy that can be shared, it is up to the cache manager to chose one to send the data. In this example, cache 1 sends its data to cache 2. All three caches now have a copy of the data in the *shared* state.

The final instruction is a write to cache 1. To gain write permissions to be allowed to modify the data, a write request is sent to the other caches. Since both caches have a valid copy of the data, they both acknowledge the request. Both caches invalidate their copies and when this is done, cache 1 is allowed to modify the data. The final result is that cache 1 has the only valid copy of the data in the *modified* state, which will eventually need to be written back to memory. These five instructions needed 11 steps due to all of the traffic that was needed to maintain coherence of data across all caches.

One optimization for this protocol is to add an additional state to reduce some unneeded coherence traffic. Consider the scenario when a block of data is loaded into cache for the first time. After the processor reads the value and wants to store a new value, an invalidation request must be sent to the other caches in the same level. However, since this copy is known to be the only copy that exists in cache, the invalidation request just added unnecessary coherence traffic. A solution to this is to add a new state, *exclusive*, that is similar to the *shared* state but is guaranteed to be the only copy in cache. Since it's the only copy, it can already have write permissions so that when a store instruction transitions it to the *modified* state, there is no need to send out an invalidation request. The only caveat is when a another cache line requests the same data, the exclusivity is lost and this cache line must be downgraded to the *shared* state. With the addition of the new state, the protocol is called MESI.

The same example is shown in Table 2.2 but this time using the MESI protocol. The difference occurs when cache 0 reads the value for the first time and then modifies it. In the MSI example, the value was read into cache 0 in the *shared* state even though it was the sole copy. In the MESI example, this same value is read into the *exclusive* state. Then, when cache 0 is going to modify the data in step 3, it can do this without issuing any other requests. The addition of this state removes the need for the unnecessary write requests when only one copy of data exists across all caches. The remainder of the example is the same as the MSI since the *exclusive* state has no impact on the other instructions.

Table 2.2: MESI Cache Coherence Example

Step	Instruction	Cache 0	Cache 1	Cache 2	Memory	Bus
0	–	(I)	(I)	(I)	3	–
1	Cache 0 Read	(I)	(I)	(I)	3	Read Request (C0)
2		3 (E)	(I)	(I)	3	Send Data (Memory)
3	Cache 0 Write	6 (M)	(I)	(I)	3	–
4	Cache 1 Read	6 (M)	(I)	(I)	3	Read Request (C1)
5		6 (S)	6 (S)	(I)	3	Send Data (C0)
6		6 (S)	6 (S)	(I)	6	Write Back (C0)
7	Cache 2 Read	6 (S)	6 (S)	(I)	6	Read Request (C2)
8		6 (S)	6 (S)	6 (S)	6	Send Data (C1)
9	Cache 1 Write	6 (S)	6 (S)	6 (S)	6	Write Request (C1)
10		(I)	9 (M)	(I)	6	–

Accessing main memory is a huge delay, whether it be reading from or writing to it. With either the MSI or MESI protocol, whenever a *modified* state gets evicted, or replaced, a write back to main memory is necessary because the *modified* state had an updated value. To avoid this, another state called *owned* is added to the protocol. This state is also similar to *shared* state, but has the sole responsibility of writing data back to memory. There can be a scenario where all caches share an updated value but main memory still has the old value. There needs to be one cache with the data in the *owned* state so there is only one responsible of updating memory when it gets evicted. Another slight optimization is that if the *owned* state gets evicted, rather than writing back the data, one of the other *shared* states can inherit the responsibility of writing back and be promoted to the *owned* state. The

protocol with the *owned* state is called MOSI. The *exclusive* and *owned* states can coexist to form the MOESI protocol. This is a very powerful and well-known cache coherence protocol for multiprocessor systems.

The coherence protocol commonly used in GPUs can be represented by an extension of MOESI. Due to the extremely high level of parallelism in GPU applications, many of the writes to the first level cache don't conflict with caches in other SMs. Since there are numerous writes that occur between all of the SMs in a GPU, the coherence traffic needed to keep all L1 cache coherent is unnecessary. By extending the MOESI protocol to allow GPUs to have some non-coherence in their L1 caches [15], the coherence bus is not saturated with unneeded traffic. The extended protocol is known as NMOESI which adds the *non-coherent* state for the first layer of cache [2].

Table 2.3 gives a brief description of six states in the NMOESI protocol. The first state, *non-coherent*, is the state added for the L1 cache of GPUs. This state allows different L1 caches to simultaneously modify the same block of data locally. The changes are not propagated to the other caches, so the changes in one L1 cache are not known to any other cache. The obvious issue with this comes when multiple SMs modify the same block of data. This process is handled when the block eventually gets written back to L2 cache. Only the data that was modified within the block gets merged with the copy in L2. The merging process is handled by a byte-mask where each bit in the mask represents each byte in the block. The value of the bit determines if the byte has been modified and requires merging [2]. This works since threads of different blocks don't usually modify the same bytes of data. In the cases that they do, atomic functions are used which don't use this *non-coherent* state and force coherency between all caches.

The second state is the *modified* state. This state signifies that changes have been made to the block and all of the other copies in the same level of cache are invalid. While there is a cache line in the *modified* state, the values in lower level memory, (i.e. closer to main memory), are outdated. Read requests for this block get the data from the *modified* cache

Table 2.3: State Definitions for NMOESI Protocol

Letter	State	Description
N	Non-coherent	Locally modified copy that other caches don't know about
M	Modified	Only valid copy and has been updated
O	Owned	One of several copies but has write back responsibility
E	Exclusive	Only valid copy of data, and it is unchanged
S	Shared	One of several copies and only as read permissions
I	Invalid	Invalid cache line, need to request data to be valid

rather than main memory so it is up to date.

Owned is the third state of the protocol. This state exists when there are multiple copies of the block across the same level of cache, but this copy owns the right to write back its data. This state allows for "dirty sharing" of the data. In the scenario when a line has modified data and another cache requests the data, instead of writing the modified data to memory, it drops from the *modified* state to the *owned* state and shares its data to the cache that requested it. This means caches can share data that main memory doesn't have an updated copy of, which reduces unnecessary accesses to main memory. The value finally gets written to main memory once the *owned* state gets evicted.

The *exclusive* state is the fourth possible state. This state signifies that this cache line has the only valid copy of the block in this level of cache. This state eases the transition to the *modified* state by not needing to send an invalidation request to other caches since this is guaranteed to be the only copy. If another cache in the the same level requests this block, the exclusivity is lost, and the cache that had the *exclusive* state must be changed to the *shared* state.

The fifth *state* is shared. When several copies exist across the same level of cache, they are in the *shared* state. If the data has been updated, one of the caches has a cache line in the *owned* state. The difference between *owned* and *shared* is that the *shared* state has no responsibilities to write back its data when it is evicted. It serves as a hit only when the cache needs to read the data.

The final state is the *invalid* state. Every other state is a valid state, meaning the data is

correct. The *invalid* state is just the opposite, the data is not correct and cannot be used. It must be overwritten and changed into one of the valid states before this cache line can have correct data.

Traditionally, all loads to cache are non-exclusive whereas stores are exclusive. Because of this, the exclusive stores are forced to be coherent which requires a lot more coherence traffic than the non-exclusive load accesses. The *non-coherent store* is a non-exclusive store meaning the access requires less coherence traffic, similar to that of a load access. This state allows for stores to the L1 cache of each SM without the overhead of keeping coherency. This allows for multiple SMs to simultaneously modify the same block in their own respective L1 caches. The blocks are then merged together with the byte-mask when written back to the lower level of cache. The merging of the data is considered safe because threads shouldn't be modifying the same bytes of data in a block. When they do, atomic functions should be used. The ability to have non-coherence applies to only the first level of cache. The lower levels do not use the *non-coherent* state which means they follow the MOESI protocol. This requires no change since it is a subset of NMOESI.

In today's GPU systems, a store instruction is assumed to be a *non-coherent store* to L1 cache. This is due to the vast number of store instructions that don't need to be coherent across all SMs of a GPU. However, there are some cases where writing data needs to be coherent, so this is handled explicitly through *atomic store* instructions. This means there are three types of memory accesses handled by the processor in GPUs: *loads*, *atomic stores*, and *non-coherent stores*.

Table 2.4 gives a brief description of the possible actions that can cause a state transition in the NMOESI protocol. The first three are processor actions. These are initiated by executing a certain instruction. The *load* is any instruction that reads from memory. The *atomic store* is an atomic write to memory. The SM that issues the *atomic store* needs to ensure that no other SM has its own locally modified copy. The final processor action is the *n-store*, or *non-coherent store*. This is issued by instructions that write to memory without

explicitly being atomic.

The next set of three actions are internal requests. The first is eviction. When a miss occurs in a cache, one cache line needs to be overwritten by the newly requested data. This is the action used to ready the line for overwriting, (i.e. invalidate and write back the data if needed). The read request is issued when a new cache line loads in data. This is to notify other caches that another copy exists in cache and to request the data so a memory access isn't needed. The write request is to notify other caches that a change is being made to one copy of the cache line. This essentially means that all other copies should send their data and be invalidated.

The final two are internal actions. These are responses to internal requests. The send data response occurs when a cache gets a read or write request. The requesting cache needs a copy of the data so this cache replies with its copy avoiding an access to main memory. Write back occurs when a block needs to be evicted to make room for a new block. The block that is getting evicted needs to store its data back to the next lower level of memory.

Table 2.4: Transition Actions in the NMOESI Protocol

Action	Description
Load	Processor executes a read from memory
Store	Processor executes an atomic write to memory
N-Store	Processor executes a write to memory
Eviction	Internal request to overwrite the cache line
Read Request	Internal request to notify other caches that another copy is going to exist
Write Request	Internal request to notify other caches that their copy is no longer valid
Send Data	Internal action to send a copy of the data to the cache that requested it
Write Back	Internal action to write data back to the next lower level of memory

Table 2.5 shows the transitions between states that occur due to the transition actions. The first column is the current state of the cache line. The next set of three columns gives the actions issued by the processor while the final set of three columns shows the internal actions to keep coherency. When the processor issues a *load* and the cache line corresponding to that address is in a valid state, then it registers as a hit and no other action is required. If the state was *invalid*, a read request is issued signifying another copy is going to exist.

Another cache in the same level may respond to the read request with the data so an access to memory isn't needed. If this occurs, the cache line transitions into the *shared* state. If a memory access is required or a lower level cache responds, the line transitions into the *exclusive* state once it receives its data. The *store* instruction always transitions the line to the *modified* state, if it wasn't already. A hit is registered if the state was already modified or had exclusive rights to modify it. Every other state must issue a write request to other caches in the same level to invalidate their copies before this cache can transition to the *modified* state. A *non-coherent store* is a little different. If the state already has write back responsibilities, (i.e. *non-coherent*, *modified*, or *owned*), then no other action is required. The other valid states must transition to the *non-coherent* state. An invalid line issues a read request to downgrade other caches' copies before transitioning to the *non-coherent* state.

When a cache line gets an eviction request, it means it must invalidate itself, if it's not already. The *non-coherent*, *modified*, and *owned* states all have the responsibility of writing back their data which means they must send their data to the lower level of memory. After they finish, then they can invalidate and be overwritten. Read requests are recognized only by states that have ownership of their data. The *non-coherent*, *shared*, and *invalid* states ignore these requests. The *owned* state responds by sending its data to the requester. The *modified* and *exclusive* state also do this but they must be downgraded to the *owned* and *shared* states, respectively, since they no longer have the sole copies of that data. Finally, a write request makes all valid copies invalid. All valid states except for *shared* respond with a copy of their data in case the requester is in the invalid state. With the write allocate policy, a write miss requires the data to be loaded into cache before overwriting it with the modified data.

These transitions are visualized in a state transition diagram in Figure 2.3. The transitions are color coded and labeled for a cleaner look. The action that caused the transition is the first label, and the optional number in parentheses is a resulting internal action caused by the transition. There are two possible transitions from the *invalid* state when a *load*

Table 2.5: State Transitions for NMOESI Protocol [2]

	Processor Actions			Internal Actions		
	Load	Store	N-Store	Eviction	Read Req.	Write Req.
N	hit	Write Req. →M	hit	Write Back →I	–	Send Data →I
M	hit	hit	hit	Write Back →I	Send Data →O	Send Data →I
O	hit	Write Req. →M	hit	Write Back →I	Send Data	Send Data →I
E	hit	hit →M	hit →N	→I	Send Data →S	Send Data →I
S	hit	Write Req. →M	hit →N	→I	–	→I
I	Read Req. →S or →E	Write Req. →M	Read Req. →N	–	–	–

is processed. These are differentiated by a single asterisk (*) or two asterisks (**). The number of copies in the same level of cache dictates whether the state transitions to *owned* or *exclusive*.

To further the understanding of the NMOESI protocol, Table 2.6 shows the cache behaviors when running a basic kernel. The function of the kernel is to add 5 to each byte of a vector. The example uses an 8-byte vector initialized with multiples 10. The kernel is designed to use 4 blocks, each with two threads. This means the first block handles the first two bytes while the last block handles the last two bytes. In this example there are three SMs which means only three blocks can be executed at a time, and the final block needs to wait for a free SM. The caches can hold one block of data, which is 4 bytes. Even though the thread block needs to access only two bytes of data, it must read data block of 4 bytes.

Table 2.6 is structured similarly to the tables of the previous cache coherence examples. The differences this time are that each cache can hold a block of 4 bytes of data. Also, there are two columns for main memory to hold the two blocks of data since the vector has a length of 8 bytes. The addresses for these two blocks are A and B, respectively. Finally, the table shows only when a change occurs. Blank cells have the same value as the cells above them. This makes identifying the changes that occur in each step easier. The example is

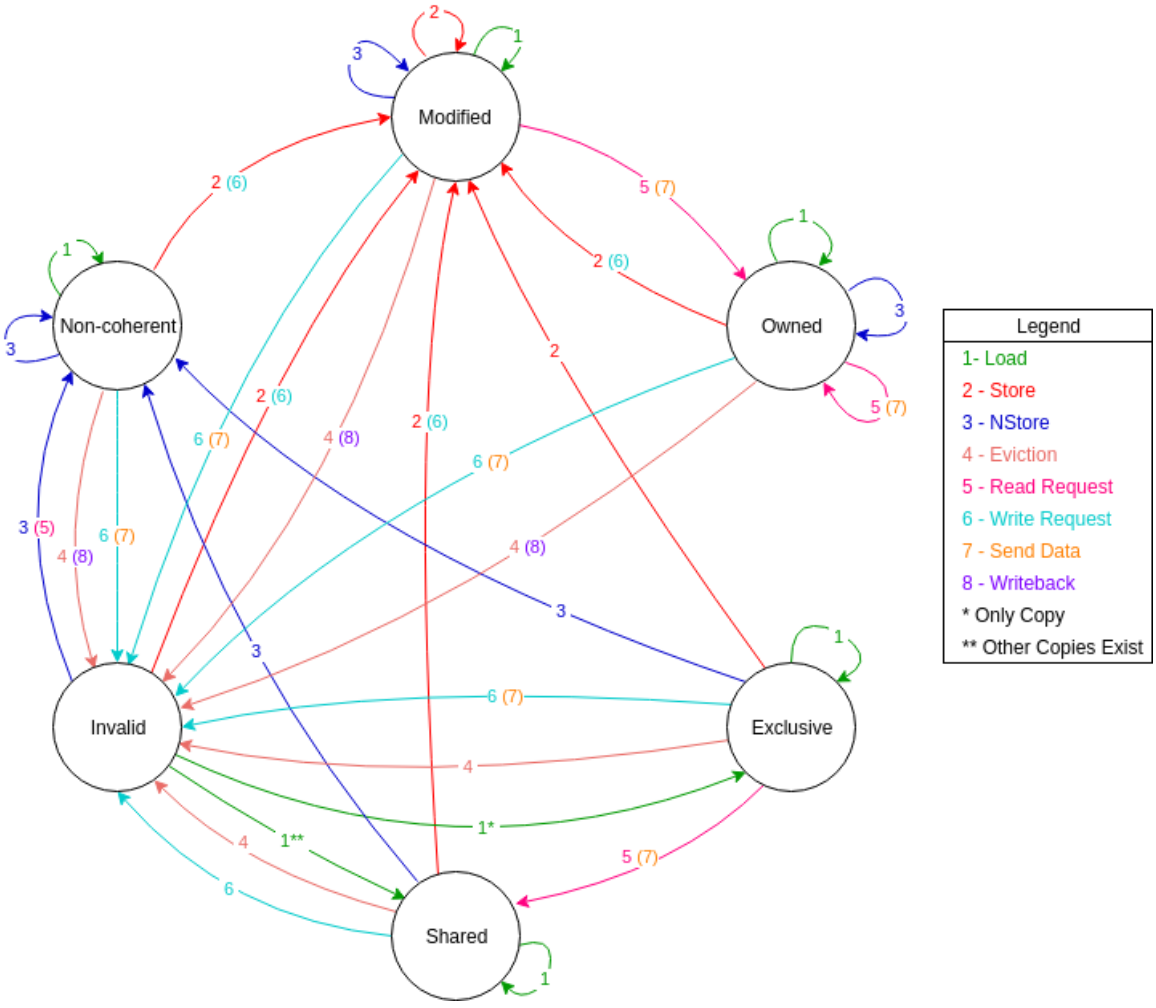


Figure 2.3: State Transition Diagram for NMOESI Cache Coherence Protocol. Transitions lines are labeled with the action that causes the transition. Actions in parentheses are additional actions that are issued after the transition.

structured in the follow way: each cache reads in the data for its thread block and modifies it non-coherently. Then, one cache needs to repeat the process for the fourth and final thread block. After that completes, the caches need to flush their modified contents to main memory so the program can complete its execution.

Step 0 shows the initialization of the program. All of the caches start in the *invalid* state and memory has the 8 bytes of the vector across two addresses. The first instruction is for cache 0 to read address A to get the first two bytes of the vector. Its first step is to issue a read request to the other caches, but since the other caches are in the *invalid* state, the request is ignored. Cache 0 must then wait for memory to send a copy of the data. Once the block of data is received, in step 2, the cache line in in the *exclusive* state since no other cache has a copy.

The second instruction is also a read to address A but for cache 1. The same address is used since this cache is handling the second thread block which uses the third and fourth bytes of the vector which is address A. A read request is issued again, but this time, cache 0 can acknowledge the request since it has a valid copy of the data. Cache 0 replies to the request by sending its copy of the data as well as transitioning its copy from the *exclusive* state to the *shared* state since its no longer the sole copy.

Cache 2 is executing the third thread block which means it needs to read address B for its data. A read request is issued to the other caches, but neither of them have a copy of address B so the request is ignored. Once the copy of data is received from memory, it is put into cache in the *exclusive* state. After these 6 steps, each cache has the data needed to execute the first three thread blocks.

Steps 7-9 are the writes to cache for the first three thread blocks. These writes transition the cache lines into the *non-coherent* states so there is no traffic on the coherence bus. This is safe to do because each thread block is accessing different bytes of data. After these three writes are complete, the execution of the first three thread blocks are also complete. This means the fourth and final thread block can be assigned to a processor and start execution.

Table 2.6: NMOESI Cache Coherence Example 2

Step	Instruction	Cache 0	Cache 1	Cache 2	Memory		Bus
					Address A	Address B	
0	–	(I)	(I)	(I)	10, 20, 30, 40	50, 60, 70, 80	–
1	Cache 0 Read A						Read Request A (C0)
2		10, 20, 30, 40 (E)					Send Data A (Memory)
3	Cache 1 Read A						Read Request A (C1)
4		10, 20, 30, 40 (S)	10, 20, 30, 40 (S)				Send Data A (C0)
5	Cache 2 Read B						Read Request B (C2)
6				50, 60, 70, 80 (E)			Send Data B (Memory)
7	Cache 0 N-Store A	15, 25, 30, 40 (N)					–
8	Cache 1 N-Store A		10, 20, 35, 45 (N)				–
9	Cache 2 N-Store B			55, 65, 70, 80 (N)			–
10	Cache 0 Read B						Eviction (C0)
11					15, 25, 30, 40		Write Back A [1100] (C0)
12		(I)					Read Request B (C0)
13		50, 60, 70, 80 (S)					Send Data B (Memory)
14	Cache 0 N-Store B	50, 60, 75, 85 (N)					–
15	End of Program Flush Cache						Write Request (Memory)
16		(I)				50, 60, 75, 85	Write Back B [0011] (C0)
17							Write Request (Memory)
18			(I)		15, 25, 35, 45		Write Back A [0011] (C1)
19							Write Request (Memory)
20				(I)		55, 65, 75, 85	Write Back B [1100] (C2)

In this example, the final thread block is assigned to cache 0. To start the execution, the cache needs to read in the bytes from address B, but the cache already has modified data loaded. This needs to be evicted before the data can be overwritten. Step 10 is the eviction request to cache 0 which cache 0 responds to by writing back its data to memory. However, since the data was in the *non-coherent* state, the write back is slightly different. Along with the data is a byte mask, shown in the square brackets. This example uses 1100 which indicated that only the first two byte have been modified, thus only those bytes should be written back to memory. This is what makes simultaneous modification of the same block of data possible. After the write back has finished, the cache line can safely be invalidated. Now the cache can issue a read request for the block of data at address B. While cache 2 does have a copy of address B, it is in the *non-coherent* state so it ignores the request. Cache 0 must wait for memory to send the data. Once it has been received, it is put into the *shared* state since cache 2 also has a copy, so it isn't exclusive.

Step 14 is the final write to finish the execution of the final thread block. At this point all of the thread blocks have finished execution. However, only one block has written its modified data to memory; the rest still exist in a cache. At the end of the program, all of the caches need to be flushed. This is a two step process for each cache. The first is a write request sent to the cache. The cache responds to this by writing its data back to memory. In step 16, cache 0 writes its data back with a byte mask of 0011 indicating that only the last two bytes should be written. After this, the cache can invalidate its copy. This process is repeated for each cache. Since the other two caches also have their data in the *non-coherent* state, they also write back data with a byte mask. The effects of this are easily shown in steps 18 and 20 when looking at the contents of memory. Only the two bytes specified by the byte mask are updated. At the end of this example, all of the caches have been flushed and the final resulting vector of 8 bytes is located in memory.

2.3 Related Work

The work by Nimkar [4] paved the way for this research. His work explored the cache access patterns of a GPU. By running various benchmarks on both NVIDIA and AMD architectures, he was able to compare cache hit ratios, most and least recently used cache lines, as well as inter- and intra-warp localities. Both architectures showed lower hit ratios than those of a CPU due to the way GPUs maintain performance compared to a CPU. GPUs balance memory misses by having a multitude of threads ready to execute whereas the CPU has a small number of threads but a much bigger cache to minimize cache misses. Nimkar conducted his research by simulating both GPU architectures with Multi2Sim. By modifying the source code, additional metrics were recorded. To measure the inter- and intra-warp locality, several counters were added to the simulator, and the counts were added to the simulation reports after running a benchmark.

The idea of modifying the cache architecture of a GPU has been proposed before. Samavatian et al. [16] propose a new type of L2 cache in order to combat the high power usage of the conventional L2 cache. To test their proposed design, a cycle accurate GPU simulator, GPGPU-Sim was used. Samavatian et al. observed that as conventional SRAM technology gets smaller, the amount of leakage current increases tenfold. This meant newer architectures used smaller cache technologies that leaked more current and thus consumed more power. They proposed switching the L2 cache to a spin torque transfer RAM (STT-RAM). This type of RAM is high density while having low leakage power. The downside to it that is has high write latency and energy. To mitigate these drawbacks, the L2 cache was split into a low retention and high retention section. The low retention section holds data that gets rewritten at a high frequency since it takes less energy to write to but doesn't retain very long. The other section takes more energy to write to it but can retain data for much longer. The findings with this new L2 cache architecture were that the power was reduced 20% while IPC was increased 16% on average. In addition to that, the size of L2

on the chip was reduced significantly since the new STT-RAM architecture is four times as dense as conventional SRAM.

Another possible modification to the cache architecture is modifying the cache coherence protocol. Candel et al. [17] used another cycle accurate GPU simulator to implement a simpler cache coherence protocol. They wanted to accurately model the memory system for GPUs for both on- and off-chip memory. To simulate the work and on-chip memory of the GPU, they used the cycle accurate simulator Multi2Sim. Another simulator, DRAMSim2, was used in conjunction with Multi2Sim to simulate the off-chip memory. The target GPU they wanted to model was an AMD Southern-Islands 7870HD. This card uses a simpler cache coherence protocol than what Multi2Sim uses, so they had to modify the source code to add this capability. The new protocol adds two bits to the instructions to determine where to access memory. The two bits are system level coherent (SLC) and global level coherent (GLC). The SLC bit indicates memory accesses should bypass cache completely and access main memory. The GLC bit is dependent and reading or writing. When reading, it means to bypass L1 and search L2. When writing, the data gets written to L1, and the GLC bit determines if the evicted line gets written back to L2. Another contrast from the NMOESI protocol is that the L1 caches use a write through policy. The L2 cache still uses write back, however. The conclusions from this paper were that the modified simulator more accurately models the AMD Southern-Islands 7870HD GPU than the original simulator. They determined this by running various performance metrics in the simulator and on the actual GPU.

Ziabari et al. [2] show that the NMOESI protocol can be used in more than just a GPU. Their work proposes a novel hardware-based unified memory hierarchy to ease memory management between a CPU and one or more GPUs. The reasoning behind using NMOESI to connect a CPU with a GPU is because the common multi-core protocol, MOESI, is a subset of NMOESI. This means all of the operations needed to keep coherency across multiple cores can be applied to CPU-GPU communication, as long as the hardware can

support it. By evaluating the design, the results showed a performance improvement of 92%, and it also improved the CPU's access to GPU's modified data by at least 13 times. In addition to all of this, performance also increased as the number of GPUs increased, proving to be a very scalable design.

Koo et al. [6] have shown that the locality of the data has significant impact on cache performance. The proposed work is a new cache manager which dynamically changes its behavior based on the locality of load instructions. Again, the work was evaluated using a simulator, GPGPU-Sim. Koo et al. observed that threads have one of four types of locality: streaming data, inter-warp locality, intra-warp locality, or a mix of both warp localities. Streaming data is data that is accessed only once. Inter-warp and intra-warp localities are when bytes of data are shared between threads from multiple warps and threads within the same warp, respectively. In addition to this, all threads in a warp share the same type of locality. A single warp is monitored to determine the type of locality and then based on this, the proposed cache manager protects the cache line fetched by the load or completely bypasses cache. To evaluate their proposed design, a wide variety of benchmarks were used across several different GPU architectures. The benchmarks varied their reliance on cache from highly sensitive to highly insensitive. The architectures included both NVIDIA's Kepler and Fermi architectures. The results from the evaluation were that the cache sensitive applications had an increase of up to 34% while the average of all was up 22%. They also report a savings of 27% in power consumption.

While GPUs have gained popularity in general purpose computing, machine learning has greatly improved by its massively parallel capabilities. Since the GPU is still designed for the general purpose, a new coprocessor called a tensor processing unit (TPU) was designed especially for machine learning. The TPU is a custom made ASIC by Google for the inference phase of neural networks. This is the prediction phase after the network has been trained. The idea behind designing this coprocessor was to remove anything unneeded from general purpose processors and focus only on the parts of the architecture that are

necessary. The architecture of the TPU is defined in [18]. There is a matrix multiply unit that takes a variable sized input Bx256. It uses a constant weights of 256x256 to multiply to get the variable output in B cycles. There is an activation pipeline that can apply the activation function, such as ReLU or Sigmoid. Also, there is dedicated hardware that can perform pooling operations. The memory hierarchy is composed of 28 MB of software managed on-chip memory and surprisingly, no cache. The idea for not having a cache is the hardware is specialized towards one specific machine learning application so there is no need for cache.

Another type of processor that can be used for machine learning is called a field-programmable gate array (FPGA). A benefit to using an FPGA is hardware acceleration. By implementing a design in hardware, it is specifically made for that application. Hardware typically runs faster than software, and there is also the added benefit of lower power consumption in hardware. Xilinx has their own deep neural network engine called xDNN [19]. These FPGA implementations outperform GPUs by having lower latency with a smaller batch sizes during the inference phase. This is because a GPU needs a large batch size since it relies on massively parallel applications. By not having to wait multiple inputs, computation can occur as soon as the first input is ready, thus reducing latency. The FPGA implementation also has the advantage of having consistent throughput throughout the entire application. GPUs have a much higher throughput but are limited by the CPU which has a much lower throughput.

Wang et al. [20] also used an FPGA because of the hardware benefits. Their work implements a deep learning accelerator unit on an FPGA. They noticed as neural networks are exploding in size, the power consumption by data centers is also increasing at an alarming rate. While GPUs can still outperform an FPGA, the power consumption of an FPGA is much less than that of a GPU while still getting moderate performance. The FPGA design is also scalable making different network topologies possible. Analysis of their implementation show that when compared to a GPU implementation, the theirs is up to ten times

more energy efficient while also consuming 364 times less power.

Chapter 3

Methodology

This work was conducted using a cycle accurate GPU simulator called Multi2Sim. This simulator features the ability to configure the architectures that it is simulating to allow for newer architectures to be emulated. The simulations were of benchmarks from two suites. One of the suites consisted of benchmarks for general purpose algorithms that are generally run on a GPU. The second is a suite of common machine learning algorithms that also benefit from being executed on a GPU. Section 3.1 goes into more detail about the simulator and how it was used. Section 3.2 explains the benchmarks used in this research.

3.1 Multi2Sim

There are two ways to explore the cache on a GPU: use a variety of GPU cards with varying cache sizes and run different analysis benchmarks on them, or use a simulator that allows for GPU architecture configuration and run those same benchmarks. The former would yield real world results at the expense of owning multiple different cards. The latter is much more feasible in terms of requiring less actual hardware and can still produce results comparable to the performance from a real GPU.

The simulator that was chosen to perform this research was Multi2Sim [21]. It was chosen over other simulator programs for a few of reasons. Among them was the ease of installation and modification as it was open source, well documented, and widely supported. Another factor in choosing Multi2Sim was the fact that it was the most up-to-date simulator

and provided the most accurate functionality for many different GPU architectures [4]. The most recent version, 5.0, was used for this research. AMD's Radeon Southern Islands and NVIDIA's Kepler architectures are the supported GPU architectures that can be emulated; however, only NVIDIA's Kepler series was targeted.

The architecture configuration and memory hierarchy were both fully configurable and could be modified for each simulation. This allowed for many different configurations to be run with relative ease. Getting results from the simulations was just as simple. On its own, Multi2Sim came with the ability to report various statistics based on how detailed of a simulation was run. These reports showed the configuration of the architecture used in the simulation and statistics about it such as core utilization among others. In addition to the architecture report, a memory report could also be generated with a detailed simulation. This report summarized the memory hierarchy as well as included cache hit and miss ratios [21]. Since Multi2Sim is open source, additional statistics could be recorded and displayed in the reports with some simple changes to the code. Nimkar [4] has done this in his work to report statistics on intra- and inter-warp hits as well as counters for the most recently used sets of the cache.

3.1.1 Kepler Architecture

Multi2Sim can model the architecture based on NVIDIA's Kepler series. The structure of the architecture is configurable, from the number of SMs, all the way down to the routing of data between specific nodes in the interconnects. To get realistic results, the architecture must be configured to mimic a real Kepler series GPU. The default memory hierarchy that Multi2Sim uses is close to the actual architecture, but there needs to be some modification in the size of the L2 cache. The default characteristics of both cache modules and the main memory module are shown in Table 3.1. It should be noted that even though the simulator uses the name Kepler for the architecture, the ability to configure the architecture allows for the emulation of features of newer architectures.

Table 3.1: Default Module Characteristics for Kepler's Memory Hierarchy

	L1 Cache	L2 Cache	Main Memory
Latency (Cycles)	6	20	300
Replacement Policy	LRU	LRU	–
Block Size (B)	128	128	128
Associativity	4 way	16 way	–
# of Sets	32	32	–
Module Size (KB)	16	64	–

Table 3.2 shows a comparison of some of the characteristics between the different generations of NVIDIA architectures. The number of SMs increases with the newer architectures after Kepler. Another difference between them all is the number of CUDA cores per SM. Kepler has the most at 192, but newer generations drop. This is because there is less overhead with fewer cores and newer technology still allowed for an increase in performance. The cache sizes have also varied with the generations. Fermi, Kepler, and Volta all have a combined L1 and shared memory. This allows for some configurability of how much to use per program. Fermi and Kepler have a total size of 64 KB while Volta has twice that at 128 KB. Maxwell and Pascal both have a dedicated 24 KB L1. Maxwell has 96 KB of shared memory while Pascal dropped this to 64 KB. Finally, the L2 cache size is an ever increasing trend. Fermi started with 768 KB, but each generation increased all the way to 6 MB with Volta. One conclusion that can be made from this table is that the cache, L2 in particular, is consuming large amounts of hardware in each new generation. Although this work focuses on the Kepler architecture, the findings and conclusions will be applicable to newer architectures. One characteristic not mentioned in the table is the associativity of the caches. While it is known that the caches use a set-associative placement policy, the number of sets and associativity are not disclosed. The only way to find this out is to benchmark the caches [10].

All of the values mentioned in Table 3.1 can be modified via a configuration file. Some of the options, such as the replacement policy, do not apply to main memory modules so the entry is just a dash. The module size is not set directly in the configuration file; rather

Table 3.2: Comparison of Various NVIDIA Architectures

	Fermi [22]	Kepler [8]	Maxwell [23]	Pascal [11]	Volta [9]
# of SMs	16	14	24	56	80
CUDA Cores/SM	32	192	128	64	64
L1 Cache Size (KB)	16 / 48	16 / 32 / 48	24	24	up to 96
L2 Cache Size	768 KB	1536 KB	2 MB	4 MB	6 MB
Shared Memory Size (KB)	48 / 16	48 / 32 / 16	96	64	up to 96

it is calculated by multiplying the number of sets, block size, and associativity together. This is the size of each cache module which is how the L1 cache size is usually specified. Since the L2 cache is shared between all SMs, its actual size needs to take into account the number of modules and multiply that by the module size.

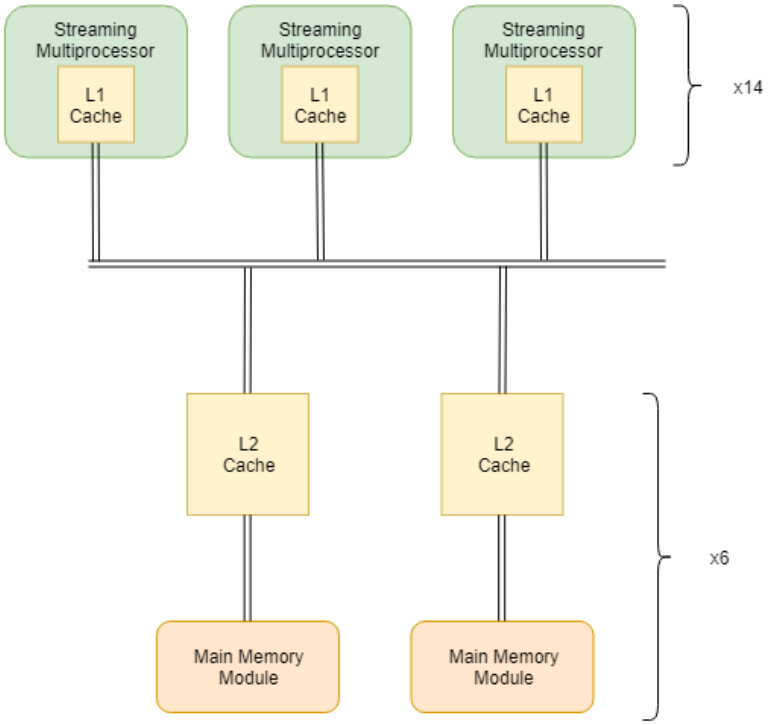


Figure 3.1: Block Diagram of the Memory Hierarchy of NVIDIA's Kepler Architecture

The structure of the hierarchy is shown in Figure 3.1. There are 14 SMs, each with their own local L1 caches. There is one network interconnect that connects each of the 14 L1 caches to each of the 6 L2 caches. Then, there is a separate interconnect that connects each

L2 cache to its own main memory module. Each main memory module serves a specific set of addresses in global memory. The two types of ways to define an address space are with an address range or by address interleaving. The simpler of these two is the address range. There is a lower and upper bound, and if an address falls between the two, that memory module serves that address. With address interleaving, the modulo operator is used on the address and the result determines which module serves it. For example, with six modules, the address would be *mod* 6 and the first module would serve all addresses that result with 0 while the last module would serve results of 5. Interleaving has a little harder calculation to do, but it splits up work much better, especially when dealing with consecutive addresses.

A separate configuration file can further modify the interconnects between each of the modules. The configuration goes as far as allowing manual routing of data between different nodes and switches. This work uses the default for the GPU configurations of just specifying the networks and using the Floyd-Warshall algorithm for routing [21]. There are a total of seven networks: one connecting all L1 modules to all L2 modules and six between each L2 module and a main memory module. All networks have three variables that are specified. The first is the default bandwidth. The configuration file specifies 264 bytes per cycle for all networks. The other two variables are the input and output buffer sizes. These are specified in number of packets and set to 528 each.

3.1.2 Limitations and Modifications

Since Multi2Sim is an actively developed open source project, there are bound to be limitations in the simulations. To simulate the CUDA framework, the functions needed to be redefined in the simulator source code. This is necessary because the functions need to work with the software defined GPU rather than looking for an actual GPU connected to the computer. Because of this, however, not all of the CUDA API function calls are implemented which limits the programs that can be simulated. For example, the function `cudaMemcpyToSymbol()` cannot be compiled because is not defined in Multi2Sim's

version of CUDA. This function is used by the CPU to initialize the constant memory of the GPU for use in the execution of a kernel. Since this function is needed to use constant memory, any program that uses it will not be able to be simulated using Multi2Sim.

Another limitation of the simulator is unrecognized instructions. The programs can be compiled, but when they are run and there is an instruction that wasn't implemented by Multi2Sim, a segmentation fault occurs. The error message says there was an unrecognized instruction at some program counter value. To determine which instruction is at that value, the CUDA binaries, called cubins, need to be debugged. The cubin file can be created using the `nvcc` compiler with the flag `--cubin`. After this, the assembly for the device code needs to be dumped. This is done using program `cuobjdump` with the flag `-sass`. For example, using the exponentiation function `expf()` produces the assembly instruction `FMUL32I`. This is a single precision floating point multiply instruction which is not supported. This is just one example of an unsupported instruction. There are more such as other floating point computations as well as type conversions that are unrecognized instructions.

There are a couple of ways to get around these limitations. The most obvious way is to implement the unsupported features as the simulator is open source. This would take a lot of additional time developing and testing the implementations. Another workaround would be to use different functions that may degrade the performance if accurate calculations are needed. However, since this research is focusing on memory usage and not so much on precision or accuracy results, another workaround was used. For example, instead of implementing the complicated calculation, such as `expf()`, a simple multiplication can be done instead. This of course is going to give incorrect results, but since the result has no effect on which memory addresses are used there will be no impact on cache performance. The ALU still takes in the same parameters, and a result is produced. The incorrect calculation acts the same from the viewpoint of the cache.

Modifications to the source code were needed to complete this research, though. Some

were fixing bugs in the code, and others were adding desired features to aid in the research. One issue that was discovered when running a long program was an unexpected crash due to a *KplStall*. This error exists when a Kepler GPU simulation doesn't complete a cycle within some threshold, signifying that is has gotten stuck. This threshold was defined to be one million. However, the way the code was written, it didn't check this. Instead it checked if the last completed cycle was greater than one million. This would trigger on any program that exceeds one million cycles, regardless of whether or not the program stalled. Looking at the AMD implementation of the same sort of error, the AMD check was correct. It used the difference between the current cycle and the last completed cycle. If this was greater than one million, then the stall error was thrown. After fixing the Kepler stall error, longer programs could safely finish.

Another modification to the code was to view the cache coherence transitions when a program was simulated. Appendix A goes into greater detail on how the simulator implements the NMOESI cache coherence protocol. A matrix of counters was added to each cache module. The size of the matrices were 6x6 for each of the possible states in the protocol. Element *ij* of the matrix represents the transition from the *i* state to the *j* state. Every time a cache line had its state set, the counter in the matrix accessed by the `from_state` and the `to_state` was incremented. At the end of program execution, this matrix would be added to the Kepler report and displayed for each cache module.

In a similar fashion, another 6x6 matrix was added to each of the cache modules. Instead of counting the number of transitions, this matrix identified the originating function call that caused the transition. The source code had 18 different instances of the `cache->setBlock()` function, which is what changes the state of a cache line. Each of these 18 instances was given unique index of a `uint` to signify that this instance was called. For example, the first instance used the least significant bit of the `uint`. If the bit was 1, that means that instance was called and contributed to the transition counts. Knowing the origin or origins of the state transitions can lead to some insight on how the cache

handles coherency.

Table 3.3 translates the origin identifier of each of the 18 calls to a brief description of when the call occurs. For example, if one entry in the matrix shows 0x4, that means the the third least significant bit is a 1. The third instance of the `cache->setBlock()` function call contributed all of the transitions for that particular entry. Looking at the translation table, this specific identifier is used when the cache module has a *non-coherent store* that causes a cache line to transition states. This method of identifying the origin of the transitions also works if there are multiple origins for a single matrix entry. If the matrix has an entry of 0xA00, that means bits 9 and 11, when counting from the left, were set. This means that two instances of the function call were responsible for the transition counts. In this example, the two were the 0x200 and 0x800 identifiers because these have only one bit set in positions 9 and 11, respectively.

Table 3.3: Coherence Matrix Origin Identifier Descriptions

Identifier	Origin
0x00001	Load Miss
0x00002	Store
0x00004	Non-coherent Store
0x00008	Eviction
0x00010	Find and Lock in Main Memory
0x00020	Evict in Main Memory
0x00040	Data Received in Evict Process
0x00080	Data Received in Non-coherent Evict Process
0x00100	Data Received in Non-coherent Evict Process
0x00200	Successful Eviction
0x00400	Write Request UpDown Finish
0x00800	Write Request DownUp Finish
0x01000	Read Request UpDown Miss
0x02000	Read Request DownUp Finish
0x04000	Read Request DownUp Finish
0x08000	Read Request DownUp Finish
0x10000	Invalidate Finish Reply with Data
0x20000	Invalidate Finish Reply with Data

3.2 Benchmarks

In order to observe the effects of GPU cache, a suite of benchmarks needs to be utilized. A benchmark suite is just collection of basic programs that each focus on a different algorithm. By using a suite, a wide variety of behaviors is executed that utilizes the cache in diverse ways. Two different suites were used in this research: a general purpose suite and a machine learning suite. The general purpose suite is a collection of a wide variety of algorithms that are suited for a GPU because of their massively parallel computation. The machine learning suite is a more targeted suite. This collection of algorithms are commonly found in many different machine learning applications that also benefit greatly when executed on a GPU. These benchmarks are not complete machine learning algorithms, only the parts of the algorithm that run on the GPU.

To ensure both benchmark suites contain valid benchmarks that execute properly, the benchmarks were run on a real NVIDIA GPU. Slight modification to the benchmarks were required to run on the actual GPU rather than the simulator. The benchmarks were designed to use Multi2Sim's version of the CUDA Runtime API. This was changed to use the version of CUDA that was installed on the Windows 10 machine, CUDA 9.0. The NVIDIA card that was used was GeForce GTX 1080.

3.2.1 General Purpose

A standard benchmark suite for NVIDIA architectures was provided with Multi2Sim. The CUDA SDK 6.5 [24] is composed of several sample programs that are targeted for GPUs. Included in the suite were programs for matrix transpose, vector addition, and fast Walsh transform, just to name a few. All of the programs had two parts: a CPU execution and a GPU execution. This allowed the program to be used as comparison between the two executions to show the speedup gained by using a GPU. Table 3.4 shows the general purpose benchmarks that were used.

Table 3.4: List of General Purpose Benchmarks

Benchmark
VectorADD
Transpose
ScalarProduction
FastWalshTransform
CppOverload
Histogram

Some modifications were made to the benchmark suite to better suit the simulator. Many of the programs used a timer module from the CUDA SDK. This would time the execution of both the CPU and GPU implementations to calculate a speedup that gets reported at the end of the program. However, this timer was not implemented in Multi2Sim, so there were warnings when simulating the program. These warnings didn't effect the execution, but produced unwanted clutter in the output so the timer was removed.

Another modification to the benchmarks was to make the CPU verification part optional. It is good to have the verification to ensure the GPU is computing the correct results, but this does add some extra time to the simulation. To keep this code optional, a `#ifdef DEBUG` block was used to encapsulate the code that runs the CPU verification. That way, the code could be compiled without the `DEBUG` flag to run without verification as well as compiled with it to include the verification.

Many of the programs had their problem sizes hard coded to a very small size, which wouldn't be enough to overflow the cache. Overflowing the cache is needed because real problems are almost always larger than the cache, and the benchmarks need to mimic that behavior. Since the size of L2 cache in Kepler architectures is 1536 KB, a size of 2 MB was used in all benchmarks for a fair comparison. All of the benchmarks were modified to take in a problem size from a parameter when running it. For the problems that dealt with vectors, this was simply just a `malloc()` call with that parameter. The matrix based problems were a little more difficult. Instead, a set of hard coded matrix sizes was available, and if the parameter sized matched, a set size of the input matrix was created. Otherwise,

a default size of 2 MB was used. The range of hard coded sizes varied from 64 KB all the way to 16 MB.

3.2.2 Machine Learning

There was not a benchmark suite for machine learning algorithms available that Multi2Sim could support until Nimkar compiled his own suite in his work [4]. This suite is of algorithms commonly found in machine learning applications. The code for them used the templates provided in the general purpose CUDA SDK [24]. Nimkar analyzed common algorithms from convolution neural network (CNN) implementations to create this suite. Specifically, the suite was created from a deep neural network (DNN) implementation in CUDA. This implementation included the basic algorithms used by most machine learning implementations such as matrix multiplication and element-wise matrix multiplication.

A common part of CNNs is the pooling layer. This layer takes a larger spatial size and downsizes it to a smaller one. There are two common algorithms that do this: mean pooling and max pooling. They work in very similar fashion, but with a slight difference in computation complexity. Mean pooling averages all of the values in the sample matrix to get a value for one element in the downsized output. This involves adding all of the elements in the sample and dividing by the total number. The sample matrix is a small subset of the input and slides around the input calculating each element of the output matrix. Max pooling is much simpler in terms of computation. Rather than finding the average of the sample, the maximum value is used. There is no division needed, and results are shown to outperform mean pooling in certain situations [25]. Sometimes this reduction in spatial size is not wanted so the matrix needs to be zero padded beforehand so the output size matches that of the original input. All zero padding does is add a boarder around the matrix of all zeros. Since this is used in some algorithms, a benchmark was created for it.

The activation function is also a key part of neural networks. It is needed to determine how to interpret the output of a neural network. There are many different activation func-

tions used, but two common ones are ReLU and sigmoid. Their functions are displayed in Equations 3.1 and 3.2, respectively. The ReLU function removes negative inputs by just setting them to zero. The positive inputs are simply passed through, unchanged. The sigmoid function bounds the output between 0 and 1. Higher positive values are closer to 1 while lower negative numbers are closer to 0. The benchmark for these two apply the respective function to an input matrix of values.

$$R(z) = max(0, z) \tag{3.1}$$

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3.2}$$

One big modification about the sigmoid benchmark is Multi2Sim does not support exponentiation on the GPU. To work around this, the actual implementation uses $e \cdot z$ in place of e^{-z}. This means the computation is incorrect, but since the output value is not used anywhere, other than the CPU verification, the memory interaction has no effect. The CPU implementation was also modified to compute this incorrect sigmoid so there can still be verification.

The final few benchmarks in the machine learning suite are for various vector element-wise operations. There are vector subtraction and multiplication benchmarks. These both take two vectors of equal length and apply the corresponding operation to every pair of elements to produce a third output vector of the same length. The scalar vector benchmark has only one input value and multiplies each element by some constant.

To expand the machine learning suite, some additional benchmarks were added that emphasized other machine learning applications. One type of neural network training is unsupervised learning. This means the data points are unlabeled, and the only way to associate bits of data together is clustering. The k-means clustering algorithm is one way of doing this. The k-means benchmark implements this algorithm. Another important

idea in machine learning is tensors. These are similar to vectors and matrices. The tensor product, or Kronecker product, is one example of a tensor operation that executes well on a GPU. The Kronecker benchmark takes an $m \times m$ and $n \times n$ matrix to produce an $mn \times mn$ matrix.

Also in the machine learning benchmark suite from [4] was an interpolation algorithm. This benchmark was intended to interpolate points within a 2D grid using bilinear interpolation. This process is more commonly found in image processing rather than machine learning, so this benchmark was not used. Table 3.5 shows the full list of machine learning benchmarks that were used.

Table 3.5: List of Machine Learning Benchmarks

Benchmark
KMeans
MaxPooling
MeanPooling
Kronecker
MatrixElemMult
ReLU
Sigmoid
VectorMul
VectorScale
VectorSub
ZeroPadding

Many of the machine learning benchmarks were modified so the structure of the code was standardized between all of the benchmarks. Another important modification was made to most of the benchmarks that operated on matrices. A lot of these benchmarks used a single dimensional block and grid. Using a 2D block and grid better fits the scope of matrices, so the benchmarks were modified to have easier thread assignment and to get better performance. Many of the changes that were made to the general purpose benchmarks were also made to the machine learning ones. The CPU verification was made optional during compilation by surrounding it in `#ifdef DEBUG` blocks. The problem sizes were all standardized to be 2 MB, for equal comparison. Finally, some of the benchmarks were

made to support a parameterized problem size. Again, if the algorithm operated on a matrix, the same range of 64 KB to 16 MB was hard coded and used if the parameter matched that size.

Chapter 4

Results and Analysis

In order to evaluate the impact of cache on performance, several experiments were conducted. The main focus was on L2 cache due to its increasingly large footprint in hardware. Preliminary research suggested a large L2 is not needed for machine learning applications. To see if this is the case, a set of benchmarks were run with varying sizes of L2 cache to compare the differences in performance. Additional experiments were conducted to support the results. These included running various problem sizes as well as recording the state transitions resulting from the cache coherence protocol.

4.1 Varying L2 Cache

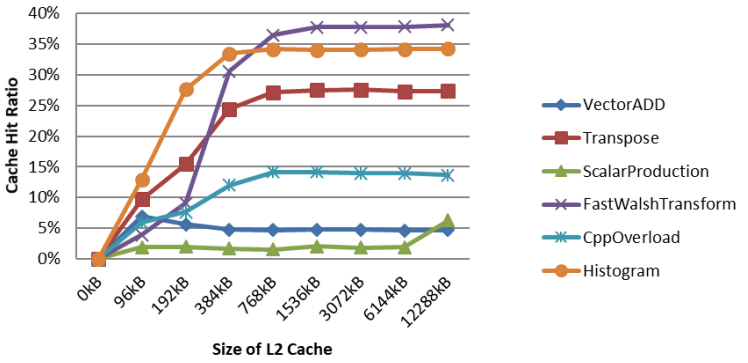
By utilizing the ability to configure the memory of the simulator, several different cache architectures were tested with both sets of benchmarks. To observe the effects the size of the L2 cache has on the performance, several cache configurations were used with varying sizes of the cache. The first point is without an L2 cache which is represented with a cache size of 0 KB. The smallest L2 size was 96 KB. By doubling the cache size repeatedly, additional sizes up to 12 MB were used. Many of the other aspects of the cache remained as the default values for the Kepler architecture. The size of the L1 cache was 16 KB. The associativity of the L1 and L2 caches were 4-way and 16-way, respectively.

To make the benchmarks simulate the real world problems, they were designed to get the best performance possible. They were organized so that the input data achieved nearly

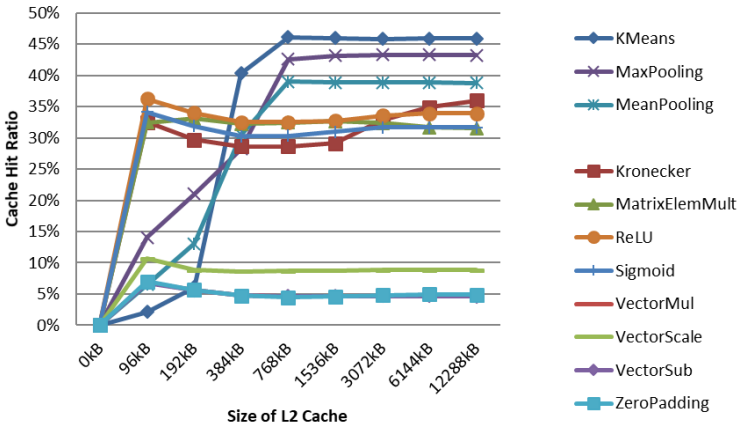
full utilization of each SM by utilizing all of its available resources. The original input size for the all of the benchmarks was hard coded to various sizes, most of which only on the order of kilobytes. To get consistent results from each benchmark, the input size for each one was set to 2 MB. This size was chosen for two main reasons. First, it is a power of 2 which makes the size of the vectors or matrices also a power of 2. This eases the thread ID calculation and makes full utilization of each SM easier. The second reason for picking an input size 2 MB was it is larger than the Kepler L2 cache size of 1536 KB. This is important because if the entire problem can fit into cache, there would never be a miss and the performance would be inflated and invalid. Actual problems that are run on a GPU are much larger than the size of the cache; otherwise there wouldn't be any benefit to running them on the GPU in the first place.

To collect the data from the simulator, a detailed Kepler simulation needed to be run. This option meant the simulator needed to record all of the metrics during the execution of the program. Additional options were specified to output the memory and architecture reports. These options to output the reports are only valid for a detailed simulation. A Python script was created to start the simulations one at a time since the simulator would crash if multiple instances of it were running simultaneously. The script ran each benchmark from both suites with all nine L2 cache size configuration files. To ensure consistent results, each configuration was run three times, and the results were averaged together since there was some variance in all of the metrics. Three metrics were recorded for comparison, and they were the hit ratios for L1 and L2 cache as well as the simulation time. The simulation time is the number of cycles the program needed to complete its execution.

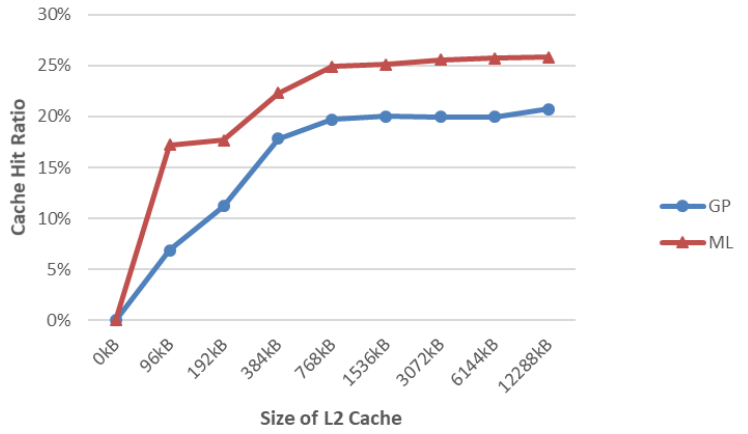
By parsing through all of the reports, the hit ratios for the L2 cache for the general purpose and machine learning benchmarks were recorded. The graphs of the hit ratios are shown in Figure 4.1. The hit ratio for L2 cache exists only when the L2 exists; hence the first data point is 0%. The trend for the general purpose benchmarks is that the performances increases with the L2 cache size until about 768 KB. After this size, the hit ratio is



(a) General Purpose



(b) Machine Learning



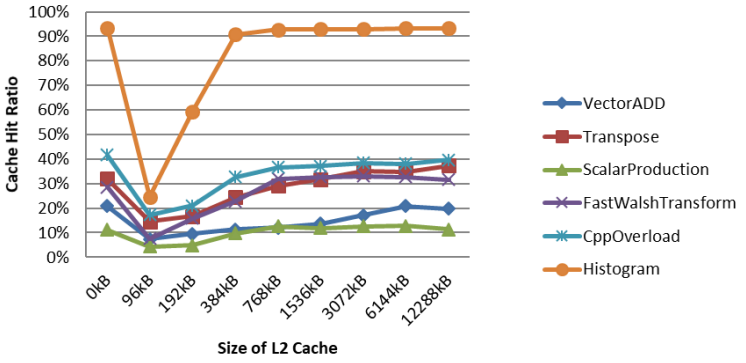
(c) Average of Both Suites

Figure 4.1: Graph of L2 hit ratios for both benchmark suites with varying L2 cache size. The size of L1 cache is 16 KB and the associativity of L2 is 16-way.

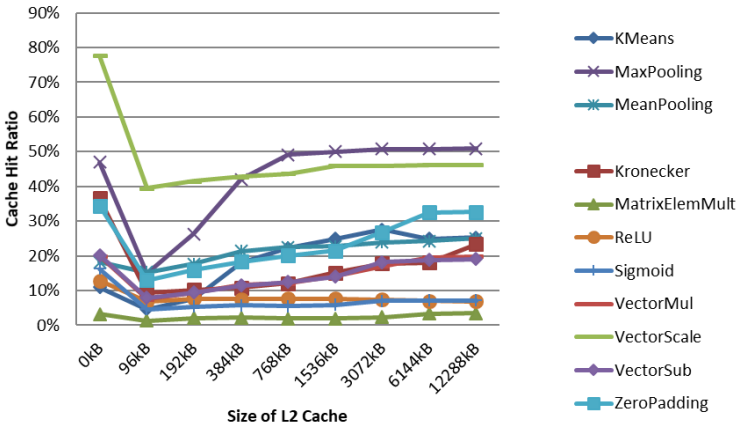
pretty consistent and a larger cache doesn't improve the performance. This is the expected behavior since a smaller cache can't hold as much data and there will be more misses, and thus a lower hit ratio. The machine learning benchmarks have two distinct trends. Some, like the pooling functions, follow the trend of the general purpose: increase up to a point and then level out. On the other hand, a majority of the benchmarks seem to be fairly constant with the size change of L2. These benchmarks are agnostic to the size of the second layer of cache. This means they perform the same regardless of how much L2 cache is available. The average of both suites of benchmarks is shown in Figure 4.1c. From this graph, it is easy to see that the machine learning benchmarks have a higher hit ratio on average for every L2 cache size when compared to the general purpose benchmarks.

The second metric that was recorded was the hit ratio of the L1 cache. This metric is important even though just the size of L2 cache was varied. The cache hit ratio is affected indirectly from this modification due to the change in coherence traffic with the different sizes of L2. The cache hit ratio of L1 is generally related to the overall performance of the application, so analysis is still beneficial. The same memory configuration was used, and the same sets of benchmarks were used. Figure 4.2 shows the graphs of the data for easier visualization. There are two main points that are made apparent by looking at the first graph, Figure 4.2a. First, the histogram benchmark averages a much higher hit ratio than any other benchmark at just over 90%. Secondly, for every benchmark, there is a large drop after introducing a small cache size. Then as the cache size is doubled, the performance improves to levels similar to the no L2 cache configuration.

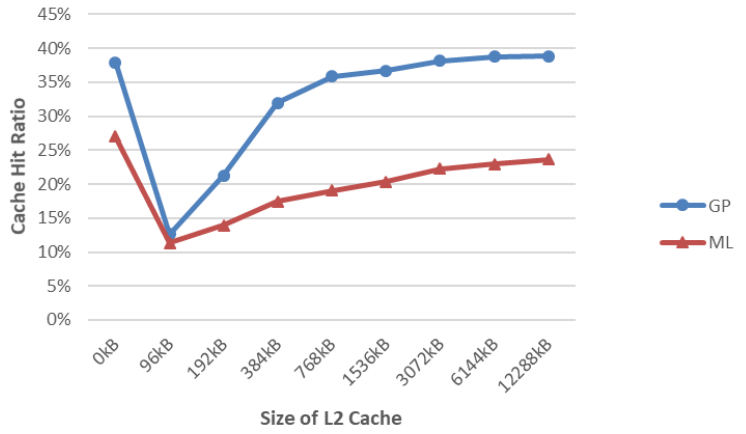
The abnormally high hit ratio for the histogram benchmark is due to its use of shared memory and how it's connected to L1 cache. The purpose of the benchmark is to sort the input data into 256 bins based on the values. In other words, the input can be thought of as a black and white image where each pixel has a value between 0 and 255. This benchmark counts how many occurrences of each pixel value that are in the image. On a CPU, the process is fairly straight forward: loop through each pixel and increment the



(a) General Purpose



(b) Machine Learning



(c) Average of Both Suites

Figure 4.2: Graph of L1 hit ratios for both benchmark suites with varying L2 cache size. The size of L1 cache is 16 KB and the associativity of L2 is 16-way.

counter corresponding to that pixel's value. This process could be implemented naively with a GPU by having each thread increment the counter corresponding to its pixel's value. Since multiple threads would be accessing the same counters, atomic functions would be necessary to ensure every thread has its value counted. However, with every thread using atomics, this becomes a sequential process and is no longer beneficial to run on a GPU.

To parallelize this process as much as possible so there is performance gain when running on a GPU, shared memory must be used. It is used so each warp in the block can work on its own sub-histogram. There is still a need for atomics but the counters are limited to each block rather than the entire device. This means threads in one block don't prevent access to threads belonging to a different block. Then, after all of the sub-histograms are computed, they can be merged together to produce the final result. This merging is done with a different kernel that has one thread per bin in the histogram. That way each of these threads is accessing its own counter, and therefore there's no need for atomics. So while atomics are useful, they can severely impact performance and should be used sparingly.

Since atomic operations are not supported in the simulator, a workaround is needed. The counters for each bin are 32 bits, and the 5 most significant bits are used as an identifier. When a thread tries to update the count, it uses it thread ID as the identifier in the counter. The thread updates the value and reads it back. If the identifier does not match, that means another thread successfully updated it. The thread tries this process again until the identifier and value match the expected identifier and value.

So how does the use of shared memory impact L1 performance? Since the shared memory is located in the same module as L1 cache, accesses to shared memory count as accesses to L1 cache. Also, the values in shared memory are always loaded so a miss never occurs. This means all accesses will count as hits and cause the hit ratio of L1 cache to be quite large.

Even with the histogram's inflated hit ratio, the trend of a small L2 cache performing worse than a large cache or no cache at all is still present. This is the case for every

benchmark. The reason for this large valley is due to the overhead with the NMOESI protocol. With a small L2, a lot of misses occur which in turn invalidate copies in L1 via write requests. A more in depth explanation is described in Section 4.2.

The second graph, Figure 4.2b, shows the hit ratios from the machine learning benchmarks. The graph is similar to the general purpose one in that the worst performance occurs with the smallest size of L2 cache. An interesting difference between the two, however, is that the performance of most of the benchmarks is greatest when the L2 cache is removed. This phenomenon is due to the high spatial locality nature of machine learning algorithms. Since accesses to memory are often sequential addresses they can be coalesced into a single access. This plus the fact the data get used within a short amount of time of being requested means the data do not need to be in L2 cache. When the cache is removed, all of the overhead needed to maintain the coherency is also removed. The benefit of having the second layer of cache doesn't outweigh the cost of the overhead to maintain the coherency. This is shown by an increase in performance when the L2 cache is removed.

The third graph, Figure 4.2c, clearly shows the difference between the average trend of both benchmark suites. The general purpose benchmarks have an overall higher hit ratio and the large spike with the removal of L2 returns to levels similar to a large L2 cache. The machine learning benchmarks are a little lower overall. The spike is higher without an L2 cache than it is with even the largest L2 size.

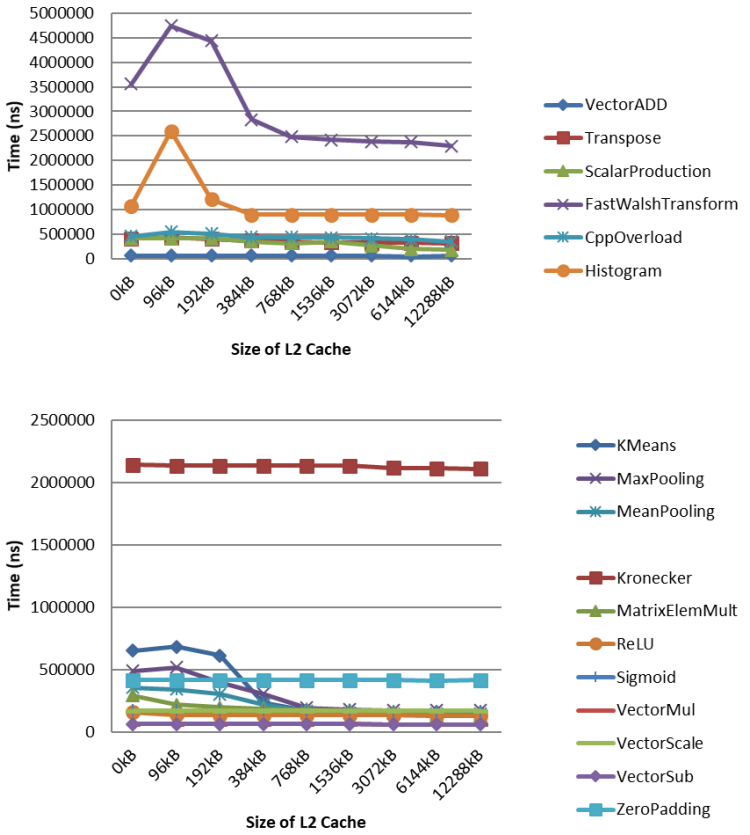
Compared to a CPU, all of the benchmarks on a GPU have a low hit ratio for both layers of cache. An average hit ratio for a cache of a CPU is around 90% [4]. The majority of the benchmarks run in this work are all less than 50%. The CPU is much more dependent on cache performing well to hide the memory accesses. This is possible since the much larger cache is shared by only a couple of threads meaning each thread gets a large portion of the available cache. The GPU on the other hand, has hundreds or thousands of threads all sharing a much smaller size cache. This is going to accelerate the replacement process in the cache which results in a lower hit ratio. This is not as detrimental to a GPU as it would

be to a CPU because the large number of threads on a GPU can hide the latency from a memory access.

The final metric for this configuration was the simulation time. This is the number of cycles needed by the program being simulated to complete execution. This is how to measure the overall performance. Figure 4.3a has the graph of the data for this metric for the general purpose benchmarks. The two benchmarks that are easy to see in the graph look like the opposite of the L1 hit ratio. Instead of a valley, there is a peak with a small L2 cache size and a sizable drop when L2 is removed. This is because the simulation time has an inverse relationship to cache performance. The better the cache performs, the lower the simulation time will be and vise versa. With the very small L2 cache, performance of the L1 cache is impacted greatly and this shows with the simulation times. Figure 4.3b shows the simulation times for the machine learning benchmarks. Many of the benchmarks look like they have a constant simulation time. The scale of the graph is dictated by the Kronecker benchmark because this has a much larger simulation time compared to the rest of them. While the simulation time is a great way to measure the overall performance of an application, it is hard to compare multiple applications because of the wide variances in times. In addition to this, averaging the simulation times will not have the desired effect since the outliers will dominate and the trend will not be valid.

A better way to compare the simulation times of multiple benchmarks is to normalize the times to get a speedup. To normalize each of the benchmarks, all of the times divide a baseline time. The baseline was chosen to be the time of the largest L2 cache size, 12288 KB. Any simulations times that are faster than the baseline will have a speedup greater than one. Conversely, times that are slower will have speedup less than one. The graphs of the speedups are shown in Figure 4.4.

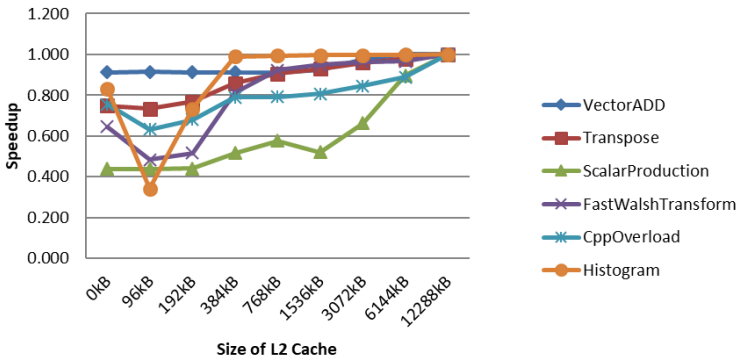
All of the general purpose benchmarks have a increasing trend with a larger L2 cache size. The machine learning benchmarks have two trends, yet again. Some of the benchmarks have a sharp increase in speedup once the cache size is large enough. However, most



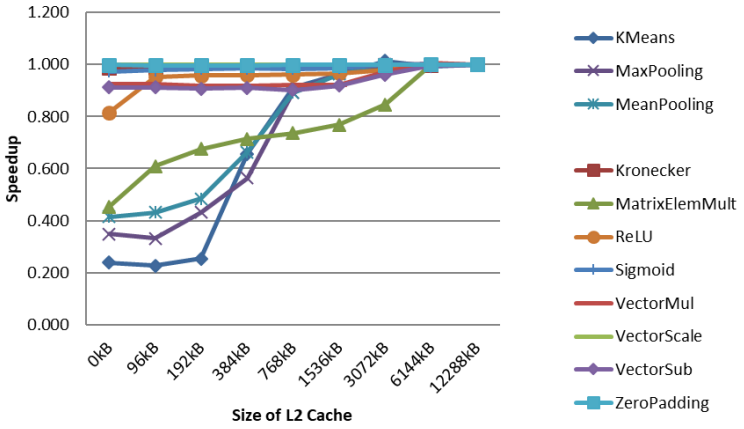
(a) General Purpose

(b) Machine Learning

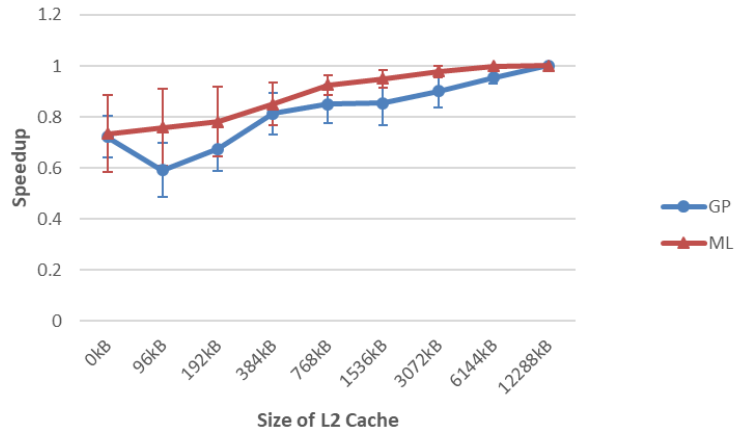
Figure 4.3: Graph of simulation times for both benchmark suites with varying L2 cache size. The size of L1 cache is 16 KB and the associativity of L2 is 16-way.



(a) General Purpose



(b) Machine Learning



(c) Average of Both Suites

Figure 4.4: Graph of simulation speedup for both benchmark suites with varying L2 cache size. The size of L1 cache is 16 KB and the associativity of L2 is 16-way.

of the benchmarks have a very slight, if any, change in speedup. The averages of the two suites are shown in Figure 4.4c. Both of the suites have similar averages since there is an increase in speedup as the size of the L2 cache increases. This is an accurate representation for the general purpose benchmarks as this is the case for all of the benchmarks. The machine learning benchmarks have two distinct trends and would be represented better with two averages. The error bars on the average speedup indicate one standard deviation away from the mean. The bars for the general purpose are closer to the average meaning the average is more representative of all of the benchmarks. The machine learning bars are wide spread meaning the set has a wide range of values.

Figure 4.5 shows the average speedups again, but there is a third average line this time. This is the average of set 2 of machine learning benchmarks. This is the majority of the machine learning benchmarks that are fairly consistent with the change in L2 size. The benchmarks that are not in the average are KMeans, MatrixElemMult, MaxPooling, and MeanPooling. These are all algorithms that have a high reuse of data. This means that every element of the input is used multiple times in the calculation of more than one output element. By reusing the data, the L2 cache is beneficial and improves overall performance. Looking the error bars for this average indicates a much better representation. They are only visible on the first point; the rest are too close to the average that the point marker is larger than the range. This indicates that the average is a much better representation of the set.

The same subset of machine learning benchmarks that have a consistent performance also share trends in the cache hit ratio metrics. The graph of the L2 hit ratio of just this subset is shown in Figure 4.6a along with the new average trend in Figure 4.6b. This average line is a good indicator of the machine learning subset. The removal of some of the benchmarks lowered the average hit ratio down from around 25% to about 18%. The new average hit ratio now correctly represents how the benchmarks are independent of the size of the L2 cache with the fairly stable trend shown by all of the benchmarks in Figure 4.6a.

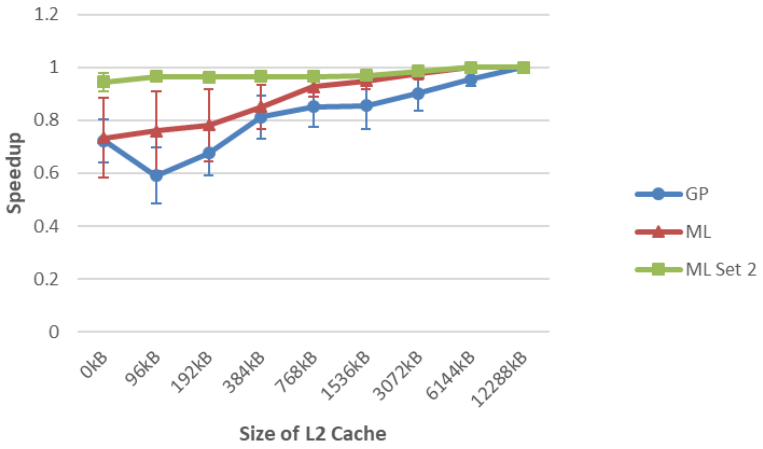
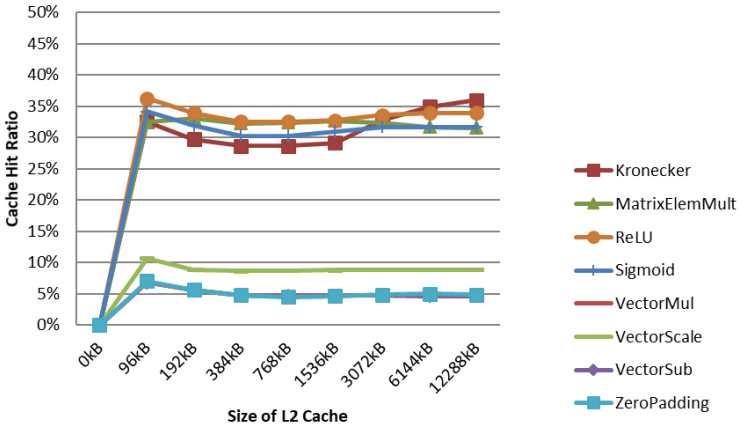
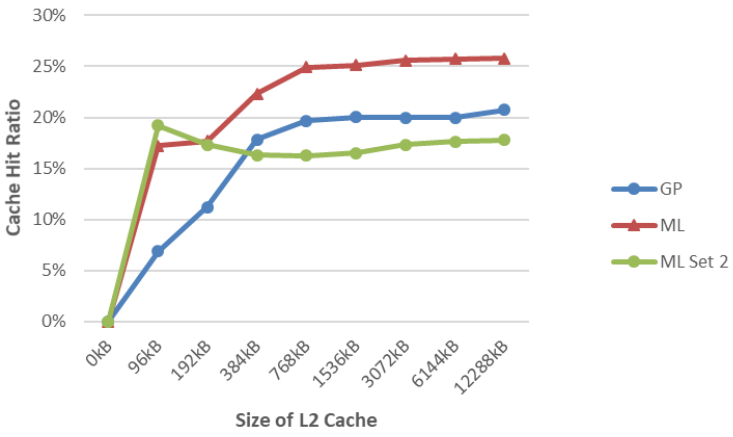


Figure 4.5: Graph of the average simulation speedup for both benchmark suites with varying L2 cache size. The size of L1 cache is 16 KB and the associativity of L2 is 16-way.



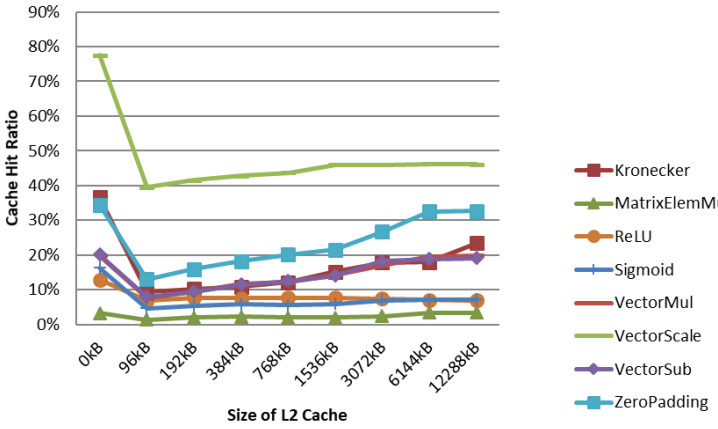
(a) Machine Learning



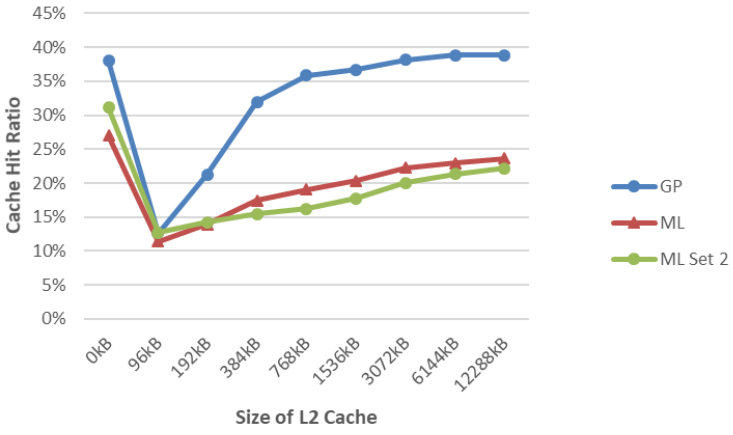
(b) Average of Both Suites

Figure 4.6: Graph of L2 hit ratios for machine learning subset with varying L2 cache size. The size of L1 cache is 16 KB and the associativity of L2 is 16-way.

Finally, the L1 hit ratios of set 2 of machine learning benchmarks are shown in Figure 4.7a. The impact on this metric is not as noticeable as the other two, but it is still present. Looking at the average L1 hit ratios in Figure 4.7b makes the impact observable. Like the L2 hit ratios, the removal of some of the benchmarks drops the hit ratio. In this case, though, the drop less than 3% for most of cache configurations. The large spike when the L2 cache is removed is different. The subset of machine learning benchmarks have a 5% higher peak. Since they have a lower average with a large cache and a larger peak without the cache, the cache performance gain is larger than the average of all of the machine learning benchmarks. As mentioned before, the better the L1 cache performs, the overall performance will also see an improvement.



(a) Machine Learning



(b) Average of Both Suites

Figure 4.7: Graph of L1 hit ratios for machine learning subset with varying L2 cache size. The size of L1 cache is 16 KB and the associativity of L2 is 16-way.

The same sets of simulations were also recorded with different cache configurations. First, the size of the L1 cache was increased to 32 KB. The results had a minor improvement in performance, but nothing too critical. The same trends were all still present in every graph of the data. A more in-depth analysis is shown in Section 4.1.1. Next, the associativity of L2 was modified. It was dropped from 16-way to 4-way. The drop in the associativity means a drop in cache size since it is used in the calculation. To compensate for the reduction, the number of sets needed to increase by the same factor of 4. This change was made to both L1 cache size configurations to make a total of four possible configurations. Again, the differences were very minor. The 16-way associative L2 cache configurations performed almost identically to the 4-way associative ones. Overall, the increase of L1 helps performance by a very minor amount, and the associativity of the L2 cache has virtually no effect on performance.

4.1.1 L1 Cache Size

As previously mentioned, the size of the L1 cache was changed and the experiments varying the L2 cache size were run again. The same 3 metrics were analyzed for each L2 cache size.

Figure 4.8 shows the average L2 hit ratios for both suites of benchmarks. The figure shows the averages of the 16 KB L1 cache configuration with solid lines and the 32 KB configuration as dashed lines. Looking at he machine learning learning benchmarks, there is no difference as the solid and dashed lines are superimposed on each other. The only difference is in the general purpose. The larger L1 size shows a lower average hit ratio for L2. This is because are there were less accesses to L2 with the larger L1 so the hit ratio is slightly lower.

The next metric is the L1 cache hit ratio. This is shown in Figure 4.9. Looking at these averages, both the general purpose and machine learning benchmarks seem to improve with the larger L1 cache size. This is expected since the large cache can hold more threads'

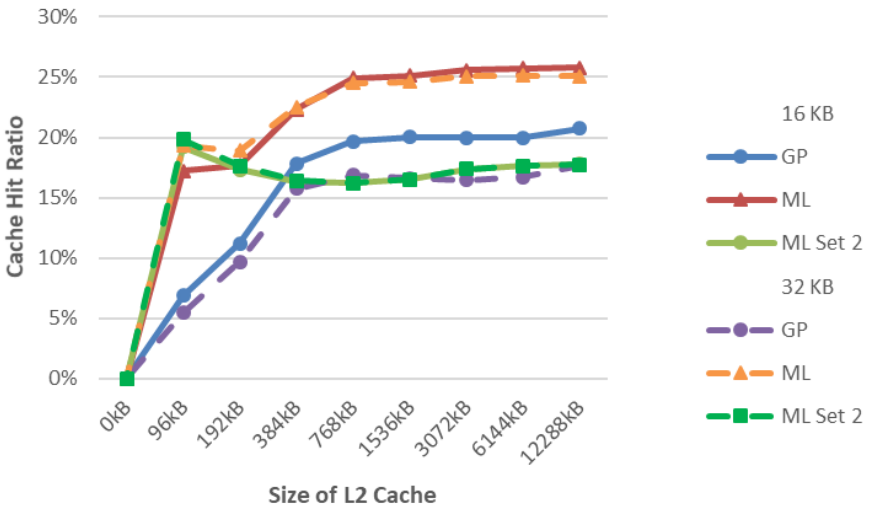


Figure 4.8: Graph of L2 hit ratios while varying L2 cache size. The size of L1 cache is 16 KB or 32 KB and the associativity of L2 is 16-way.

data, there is less misses. Set 2 of the machine learning benchmarks don't show that much of a difference, however. This suggests that a majority of the benchmarks do not see a performance increase in the L1 cache when it is doubled in size.

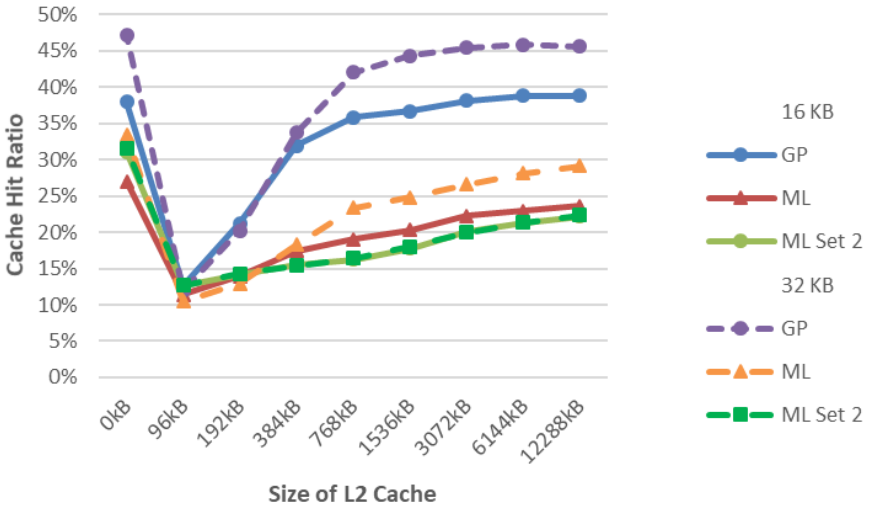


Figure 4.9: Graph of L1 hit ratios while varying L2 cache size. The size of L1 cache is 16 KB or 32 KB and the associativity of L2 is 16-way.

The final metric, shown in Figure 4.10, is the simulation speedup. This graph shows that all three trends are the same with both sizes of L1. The size of the L1 cache has little

effect on the varying sizes of the L2 cache. However, when comparing the simulations times of 16 KB of L1 and 32 KB of L1, the larger size has slightly faster times. Again, this is expected as the larger cache is saving on memory accesses, reducing the total number of cycles needed for the program to complete.

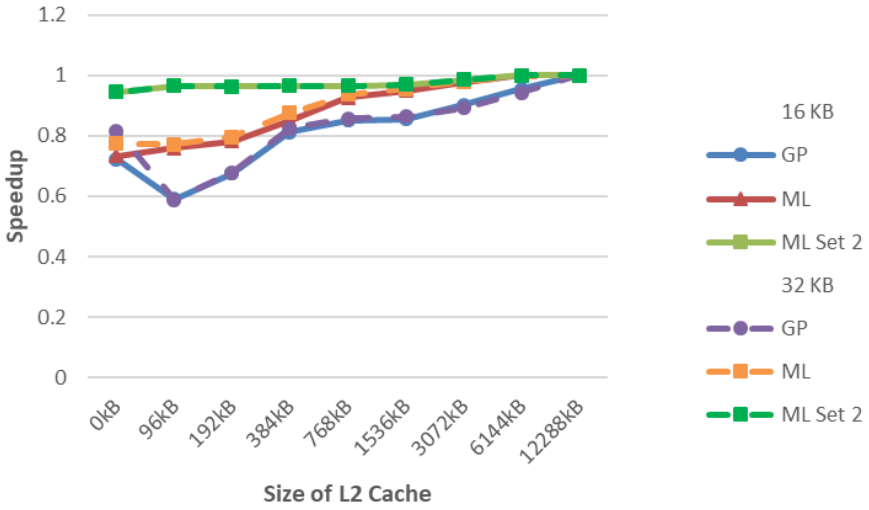


Figure 4.10: Graph of simulation speedup while varying L2 cache size. The size of L1 cache is 16 KB or 32 KB and the associativity of L2 is 16-way.

4.2 Coherence Matrices

To determine the effect of the NMOESI protocol on the cache, two sets of cache coherence matrices were added to the simulator. The first counted the number of transitions between each pair of states in the protocol, referred to as the transition count matrices. The second tracked the origin of the transitions to the event that issued the change by using a unique identifier. These are known as the origin identifier matrices. Table 4.1 is a copy of the origin identifier descriptions that are used in these origin identifier matrices. By analyzing these matrices, the reason for the valley in the L1 performance could be explained.

Each benchmark from both suites were run with three different cache configurations to get the coherence matrices statistics. The three configurations each had a 16 KB L1 cache and a 16-way associative L2 cache. As shown by the previous results, the size of the L1

Table 4.1: Coherence Matrix Origin Identifier Descriptions

Identifier	Origin
0x00001	Load Miss
0x00002	Store
0x00004	Non-coherent Store
0x00008	Eviction
0x00010	Find and Lock in Main Memory
0x00020	Evict in Main Memory
0x00040	Data Received in Evict Process
0x00080	Data Received in Non-coherent Evict Process
0x00100	Data Received in Non-coherent Evict Process
0x00200	Successful Eviction
0x00400	Write Request UpDown Finish
0x00800	Write Request DownUp Finish
0x01000	Read Request UpDown Miss
0x02000	Read Request DownUp Finish
0x04000	Read Request DownUp Finish
0x08000	Read Request DownUp Finish
0x10000	Invalidate Finish Reply with Data
0x20000	Invalidate Finish Reply with Data

cache and the associativity of the L2 cache have minimal effect of the metrics, so this part of the configuration was chosen due to the fact it was the Kepler default. The difference between the three was in the size of L2. This was either 1536 KB, 96 KB, or 0 KB (no L2 cache). These three sizes were enough to replicate the valley in the L1 hit ratio. Looking at the transition count matrix of the L1 cache for each of the benchmarks, almost all of them show the same trend across the three configurations. There are variations in the counts of each transition between the different benchmarks, but comparing the relationship of the three configurations, they are almost all identical. The origin identifier matrices between all the benchmarks are identical.

Table 4.2 shows the two sets of cache coherence matrices for the transpose benchmark. This benchmark was chosen arbitrarily since most of them show basically the same trends. Table 4.2a gives the counts of the transitions that occurred in the L1 cache for each size of L2. For each matrix, element *ij* is number of transitions from state *i* into state *j*. For

example, looking at the first row of the first matrix in Table 4.2a, the only non-zero counts are in the *N* and *I* columns. This means that whenever a cache line was in the *non-coherent* state, it was either updated and remained in that state or it was invalidated and transitioned to the *invalid* state. This is the case for the other two sizes of L2 as well, but the 96 KB L2 has a slightly higher count of *non-coherent* to *invalid* transitions.

Table 4.2b holds the matrices of origin identifiers. The matrix is laid out in the same manner as the transition counts matrix. Looking at the first row of the first matrix again, it shows the same two columns have non-zero entries. In fact, for every non-zero entry in the count matrices, the corresponding entry in the origin identifier matrix is also non-zero. Each identifier in these matrices is a hexadecimal number. For clarity, the 0x prefix and the leading zeros have been dropped. Table 4.1 gives a brief description of where each identifier is called in the event hierarchy. The entry for the *non-coherent* to *non-coherent* state transition is 0x4. This identifier means the transition comes from a *non-coherent store* instruction. What this transition means is a copy of the data already existed in L1 cache when the SM processed a write instruction. The cache line registered a hit, and the block of data was updated. The identifier for the *non-coherent* to *invalid* transition is 0x200. This identifier is called when an eviction is successful. A successful eviction is when the block of data that is being evicted was successfully written back and is able to be overwritten by the new block of data that missed in cache.

One oddity about the count matrices is that the rows and columns for the *modified* and *owned* states are all zero. The reason for this is because the only way to transition to the *modified* state in L1 is via an *atomic store* instruction. None of the benchmarks, except for the histogram, use atomic functions because there is no need for them. Since the use of atomics hurts the parallel performance, it is best to design the benchmarks without them. The histogram benchmark can be designed to use atomics, but since the simulator doesn't support this, there are no *modified* states for the L1 cache even for this benchmark. It follows that the reason there are no *owned* state transitions is because the only way to be in

Table 4.2: Cache Coherence Matrices for L1 Cache from Transpose Benchmark

(a) Transition Count Matrices

		N	M	O	E	S	I
1536 KB L2	N	109	0	0	0	0	1323
	M	0	0	0	0	0	0
	O	0	0	0	0	0	0
	E	172	0	0	0	957	4148
	S	0	0	0	0	0	1922
	I	1161	0	0	5333	1027	7393

		N	M	O	E	S	I
96 KB L2	N	109	0	0	0	0	1663
	M	0	0	0	0	0	0
	O	0	0	0	0	0	0
	E	126	0	0	0	475	4213
	S	0	0	0	0	0	754
	I	1544	0	0	4824	285	2518

		N	M	O	E	S	I
0 KB L2	N	101	0	0	0	0	1369
	M	0	0	0	0	0	0
	O	0	0	0	0	0	0
	E	220	0	0	0	928	2866
	S	0	0	0	0	0	1871
	I	1185	0	0	4069	980	6106

(b) Origin Identifier Matrices (Hex)

		N	M	O	E	S	I
1536 KB L2	N	4	0	0	0	0	200
	M	0	0	0	0	0	0
	O	0	0	0	0	0	0
	E	4	0	0	0	8000	200
	S	0	0	0	0	0	200
	I	4	0	0	1	1	8

		N	M	O	E	S	I
96 KB L2	N	4	0	0	0	0	A00
	M	0	0	0	0	0	0
	O	0	0	0	0	0	0
	E	4	0	0	0	8000	A00
	S	0	0	0	0	0	A00
	I	4	0	0	1	1	8

		N	M	O	E	S	I
0 KB L2	N	4	0	0	0	0	200
	M	0	0	0	0	0	0
	O	0	0	0	0	0	0
	E	4	0	0	0	8000	200
	S	0	0	0	0	0	200
	I	4	0	0	1	1	8

that state is to come from a *modified* state. The state diagram in Figure 2.3 makes this very clear. Other than giving an idea of how often each transition occurs, the counts for each transition don't easily explain why the valley occurs in the L1 hit ratio. For this, the origin of each transition also needs to be taken into consideration.

Comparing the origins of the transitions for each of the three L2 sizes, they are almost all identical expect for 3 entries all in the *invalid* column. For the 1536 KB and 0 KB matrices, these three entries all have the identifier that means a successful eviction, 0x200. The small L2 cache of 96 KB is a little different. It has the identifier of 0xA00. This identifier isn't in the table of origin descriptions because it is actually two identifiers: 0x200 and 0x800. What this means is that some of the transitions were caused by a successful eviction (0x200) while others were caused by a write request downup finish (0x800). This second transition origin is the reason why the size of the L2 cache hurts the performance of the L1 cache.

The first part of the origin description is the write request. As described in Table 2.4,

a write request is an internal request to notify caches that their current copy of the data is now invalid. This tells the cache to transition the cache line that had the data into the *invalid* state. The cause of a write request is explained in Table 2.5. The processor action *store* is the only way a write request is issued. Since atomic stores are not supported in the simulator, this *store* action had to come from the L2 cache.

This is confirmed by the second part of the origin description, downup. This describes the direction of the write request: from down to up, or in other words, from the lower memory to high memory. Lower memory is closer the main memory while higher memory is closer to the processor. This means the write request was sent by L2 to invalid copies in L1. The reason this is dependent on the size of L2 is because it is issued only when a miss occurs. With a small L2, misses are going to happen with a much higher frequency and in turn, more write requests are going to be issued. This request invalidates all copies in all L1 caches which ultimately causes more misses to occur in L1, greatly hurting performance. The reason this doesn't occur when there is no L2 is because the main memory will never have a miss, meaning no write requests will be issued. The final part of the origin, finish, just signifies that the write request finished successfully.

The only benchmark that had a different trend from these L1 coherence matrices was the histogram benchmark. The reason for this is yet again due to the heavy use of shared memory. Table 4.3 has the matrices for the benchmark. Comparing this table to the previous one shows the only differences are with the transition counts. The origin identifiers are exactly the same. An easy difference to notice in the transition count matrices is the number of transitions into the *non-coherent* state. The histogram benchmark has a lot more write accesses due to the atomic addition workaround, and they are evident with this large number of transitions. Another discrepancy is the number of transitions into *invalid* is high with the small cache compared to the other two configurations. Because of the high number of writes and the small L2 cache constantly invalidating the copies in L1, it follows that there will be a high number of invalidations.

Table 4.3: Cache Coherence Matrices for L1 Cache from Histogram Benchmark

(a) Coherence Count Matrices

1536 KB L2	N	M	O	E	S	I
N	30914	0	0	0	0	2412
M	0	0	0	0	0	0
O	0	0	0	0	0	0
E	1883	0	0	0	829	5548
S	0	0	0	0	0	2979
I	539	0	0	8306	2221	10939

96 KB L2	N	M	O	E	S	I
N	22631	0	0	0	0	10914
M	0	0	0	0	0	0
O	0	0	0	0	0	0
E	6279	0	0	0	367	63205
S	0	0	0	0	0	853
I	4636	0	0	69884	491	19103

0 KB L2	N	M	O	E	S	I
N	25076	0	0	0	0	2107
M	0	0	0	0	0	0
O	0	0	0	0	0	0
E	1631	0	0	0	906	4679
S	0	0	0	0	0	2793
I	497	0	0	7225	1984	9579

(b) Origin Identifier Matrices (Hex)

1536 KB L2	N	M	O	E	S	I
N	4	0	0	0	0	200
M	0	0	0	0	0	0
O	0	0	0	0	0	0
E	4	0	0	0	8000	200
S	0	0	0	0	0	200
I	4	0	0	1	1	8

96 KB L2	N	M	O	E	S	I
N	4	0	0	0	0	A00
M	0	0	0	0	0	0
O	0	0	0	0	0	0
E	4	0	0	0	8000	A00
S	0	0	0	0	0	A00
I	4	0	0	1	1	8

0 KB L2	N	M	O	E	S	I
N	4	0	0	0	0	200
M	0	0	0	0	0	0
O	0	0	0	0	0	0
E	4	0	0	0	8000	200
S	0	0	0	0	0	200
I	4	0	0	1	1	8

The same coherence matrices were recorded for the L2 cache modules. To support the notion of L2 sending write requests, these matrices should have some transitions into the *modified* state. Table 4.4 holds these matrices from the transpose benchmark. Again, this was chosen arbitrarily since all of the benchmarks produced similar matrices with slight variances in the transition counts. There are only two matrices per table this time since there isn't an L2 cache to report these metrics in the third configuration. Looking at the counts of the transitions in Table 4.4a, the second configuration with the smaller L2 cache has higher counts for every entry. This means that there were more accesses to the module in order to maintain the coherence. Another thing to note is that the row and column for the *non-coherent* state are always zero. This is because that state is valid for only the L1 cache because the L2 caches must always maintain coherence with each other. Similarly, the rows and columns for the *owned* and *shared* states are also all zero. These are also not valid states for the L2 cache in this specific cache configuration. The way the L2 cache is configured, each module handles accesses to its own set of addresses. This means that there cannot be multiple copies of the same block of data across multiple L2 caches, which is what the *owned* and *shared* states are for.

Table 4.4b has the origin identifier matrices. The matrices for the 1536 KB L2 and the 96 KB L2 are identical except for the *modified* row. This is expected based on the information gained from the L1 coherence matrices, the difference being the addition of the identifier 0x10000. According to Table 4.1, this identifier is for an invalidate finish reply with data. What this means is that the write request that was sent out by the L2 cache successfully invalidated copies in higher cache (i.e. L1 cache). In addition to that, the L1 cache also responded with data that the L2 cache needs to store. This is why the cache line transitioned into the *modified* state.

Table 4.4: Cache Coherence Matrices for L2 Cache from Transpose Benchmark

(a) Coherence Count Matrices

1536 KB L2		N	M	O	E	S	I
	N	0	0	0	0	0	0
	M	0	0	0	0	0	2516
	O	0	0	0	0	0	0
	E	0	3045	0	0	0	3549
	S	0	0	0	0	0	0
	I	0	0	0	8109	0	6065

96 KB L2		N	M	O	E	S	I
	N	0	0	0	0	0	0
	M	0	8	0	0	0	3962
	O	0	0	0	0	0	0
	E	0	4001	0	0	0	10222
	S	0	0	0	0	0	0
	I	0	0	0	14312	0	14184

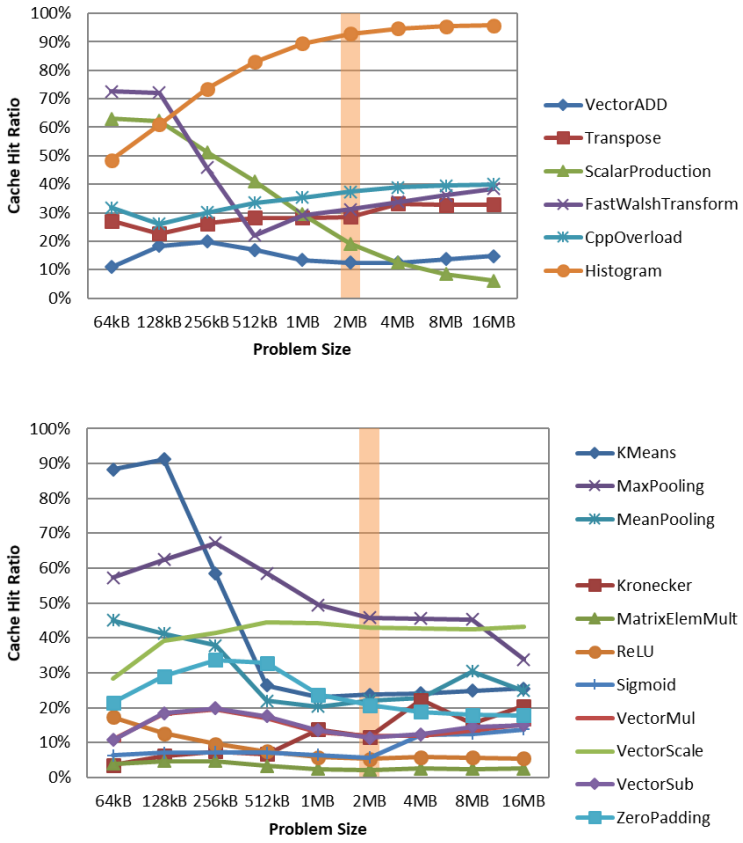
(b) Origin Identifier Matrices (Hex)

1536 KB L2		N	M	O	E	S	I
	N	0	0	0	0	0	0
	M	0	0	0	0	0	200
	O	0	0	0	0	0	0
	E	0	80	0	0	0	200
	S	0	0	0	0	0	0
	I	0	0	0	1000	0	8

96 KB L2		N	M	O	E	S	I
	N	0	0	0	0	0	0
	M	0	10000	0	0	0	200
	O	0	0	0	0	0	0
	E	0	10080	0	0	0	200
	S	0	0	0	0	0	0
	I	0	0	0	1000	0	8

4.3 Varying Input Data Size

The size of the input problems for the previous results were all constant at 2 MB for each benchmark. To gain a perspective on the effects that the problem size has on the cache performance, all of the benchmarks were run with a variable input size. The range was from 64 KB to 16 MB. All of the simulations used the same cache configuration: 16 KB of L1 and a 16-way associative, 1536 KB L2 cache. All of the figures have a shaded column representing the baseline size of 2 MB used in the previous simulations. The L1 hit ratios for for both sets of benchmarks are shown in Figure 4.11.

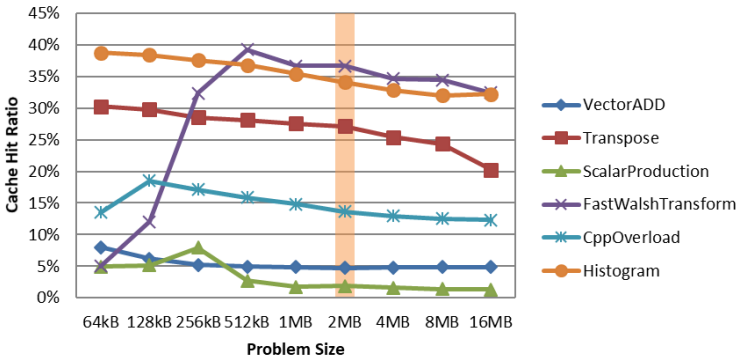


(a) General Purpose

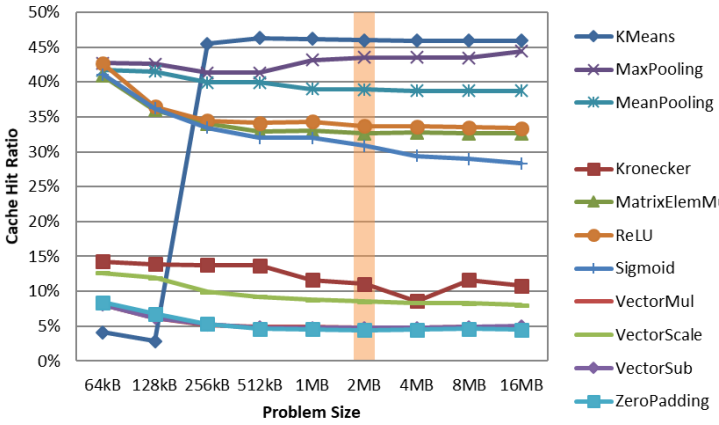
(b) Machine Learning

Figure 4.11: Graphs of the L1 Hit Ratios for Both Benchmark Suites

Figure 4.11a shows the hit ratios for the general purpose benchmarks. The Histogram benchmark shows a large increase in performance with the larger problem sizes while the others show just a slight increase. This is due to the fact that the histogram has a high L1 performance due to shared memory. The large increase is because the same 256 bins are in shared memory and the larger input size leads to more hits when incrementing the counters. Figure 4.11b shows the hit ratios for the machine learning benchmarks. Looking at the smaller input sizes, there are a lot more fluctuations. This is because the problem size is too small for the cache performance to settle out. With the larger sizes, this is not the case. The performance seems to stable out for almost every benchmark.



(a) General Purpose

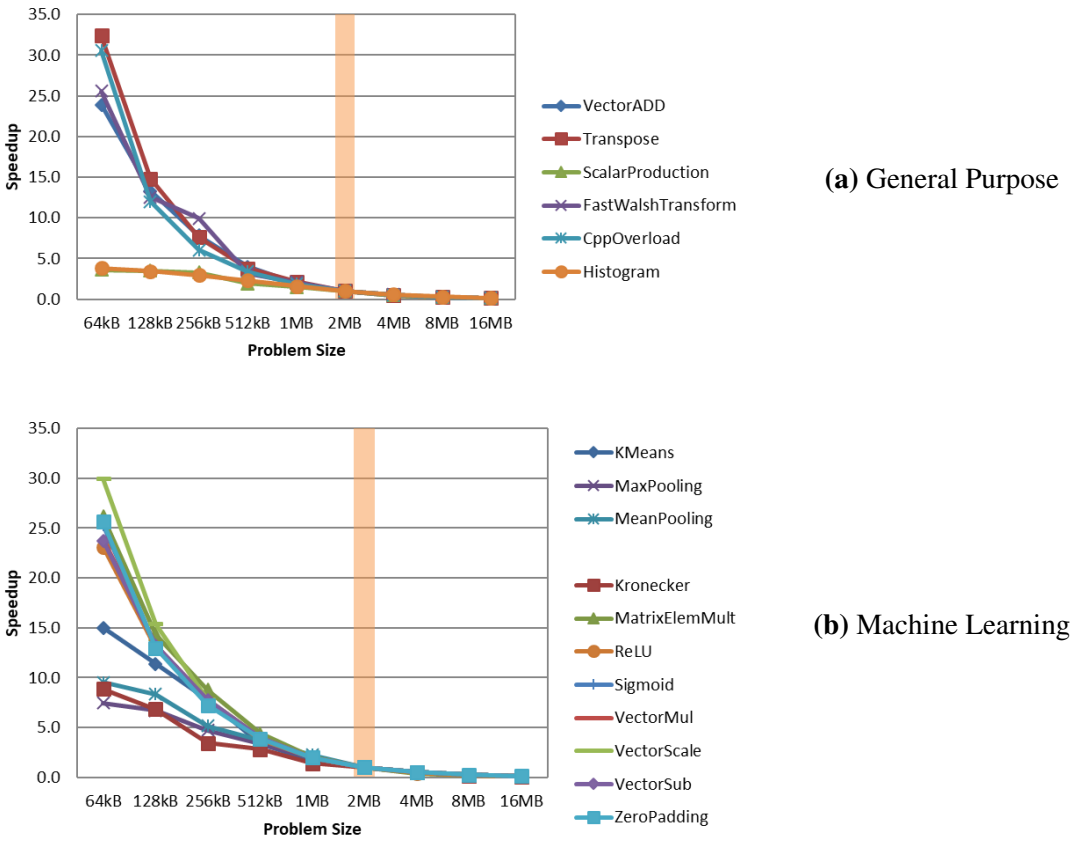


(b) Machine Learning

Figure 4.12: Graphs of the L2 Hit Ratios for Both Benchmark Suites

The next set of graphs in Figure 4.12 are for the L2 cache performance. Again, the general purpose benchmarks are the first graph, Figure 4.12a. The trends for all the benchmarks are very similar, except for the smaller sizes. As the problem size increases, they each drop in performance at a very slight rate. This behavior is to be expected as the blocks of data in the cache need to be replaced more often. The smaller input sizes are not large enough for the performance to settle. The machine learning benchmarks in Figure 4.12b show a slightly different trend, however. As the problem size increases, the performance of the L2 cache seems to steady out. This suggests that the spatial locality of the data in machine learning benchmarks has a significant impact on cache performance. When a cache miss occurs, the surrounding bytes of data in the block that get written to cache are also used later, making up for the miss. In other words, when one thread misses cache, the block of data that is retrieved from memory is used by other threads. This makes the cache

performance almost independent from problem size. A more in depth experiment of spatial locality on cache performance is shown in Section 4.4.



(a) General Purpose

(b) Machine Learning

Figure 4.13: Graphs of the Simulation Times for Both Benchmark Suites

The final metric, speedup, is shown in Figure 4.13. The baseline is with a 2 MB input size. The general purpose benchmarks all have the same exponential decay, with slight variances between them. This is shown in Figure 4.13a. The machine learning benchmark trends are almost identical to the general purpose ones. The trend of the lines in Figure 4.13b are also exponential with the varying input size. This is due to the fact that all the benchmarks are designed for use on a GPU, and this is the natural behavior of increasing the problem size for the programs. Aside from the smaller problem sizes, the global trend is as the size is doubled, the speedup is cut in half.

4.4 Spatial Locality Benchmark

To verify the effect that spatial locality has on performance, a dummy benchmark was created. This benchmark was designed to have poor hit ratio performance by accessing random memory. The task of the kernel was very simple. All it did was copy one element from the input matrix to the output matrix. The index of the input matrix was chosen to be random, or as random as can be with the limitations of the simulator. The ideal GPU program has threads of a warp accessing sequential data so the accesses can be coalesced into a single access to memory. By using random indices, no thread access could be coalesced with others in the same warp leading to a very poor hit ratio. The way the random index was computed was by bit reversing the current thread ID. The number of elements in the input matrix was always a power of 2 which eased this process. For example, if the input was a 256x256 matrix, the thread IDs would need 16 bits since there are a total of 65536 elements. The reverse of thread ID 10, 0x000A in hexadecimal, would be 20480 or 0x5000. The benefit of this approach is that the next thread ID, 11 or 0x000B reverses to 53248 or 0xD000. This is a difference of 32768, much too large to ever get coalesced. This process isn't completely random as it will produce the same computations every time it runs. However, the randomness that occurs between sequential thread IDs is what the goal really was so they couldn't be coalesced.

To get a complete view of the effect of spatial locality, a wide variety of memory configurations and problem sizes were simulated. The input data size was varied from 64 KB to 16 MB. The size of the L2 cache varied between no cache and 1536 KB. The associativity was either 4-way or 16-way. The same three metrics were recorded: hit ratios for L1 and L2 cache as well as the simulation time. Figure 4.14 shows a graph of the L2 hit ratios. The x-axis shows the increase in the input data size. Each line represents a different cache configuration for the L2 cache. For consistency, the line for no cache is still shown as % for all problem sizes.

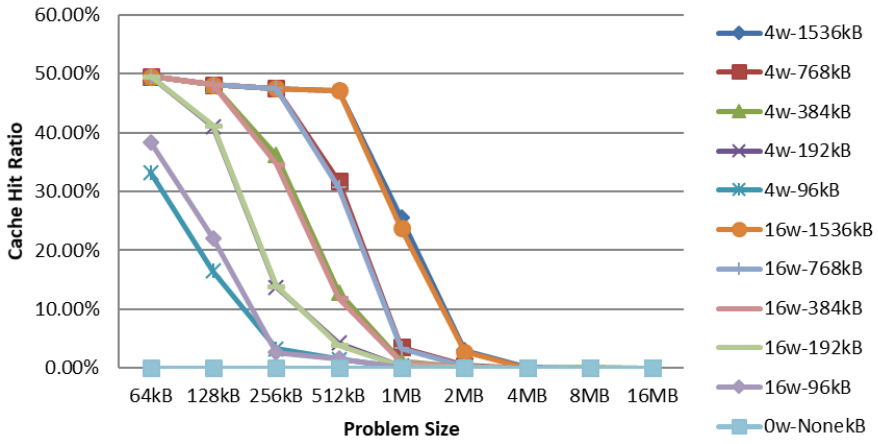


Figure 4.14: Graph of L2 Hit Ratio for Spatial Locality Benchmark. The size of L1 cache is 16 KB.

There is a clearly defined relationship between the problem size and the size of L2. When the size of the cache can easily hold the entire problem, the hit ratio is near 50%. However, once the problem size approaches the size of cache, the hit ratio begins to degrade. After the problem size exceeds the cache size, the performance drops to virtually 0%. This may seem like it could be alright, but the largest size simulated, 16 MB, is not even close to the size of an actual problem that would benefit from executing on a GPU. These small benchmark sizes are enough to get characteristics of the cache performance, but not enough to show any speedup when compared to a CPU's execution of them. Another thing to note is that the associativity still doesn't have a large impact on the performance. The points for the 4-way and 16-way associativity are nearly identical for each cache size.

Figure 4.15 shows a graph of the L1 hit ratios. From looking at the graph, it seems erratic with no hint of a pattern. However, looking at the scale shows that they are all less than 1.5%. Since this is such a small magnitude, even a little change in the number of hits or misses looks like a steep slope. This is the case because the all of the problem sizes are much larger than the L1 cache size. This is how the L2 hit ratio graph would like if only the points for 4 MB to 16 MB were graphed.

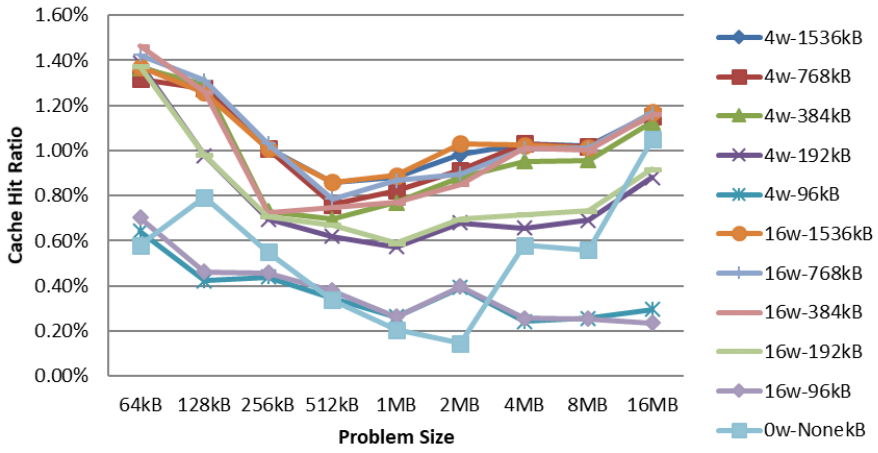


Figure 4.15: Graph of L1 Hit Ratio for Spatial Locality Benchmark. The size of L1 cache is 16 KB.

The final metric that shows the effect of spatial locality is the simulation time, the number of cycles that the simulator needed to complete the execution of the program. The same set of simulations was used, and the simulation times recorded. The graph in Figure 4.16 shows the effect of the larger problem sizes on each cache configuration.

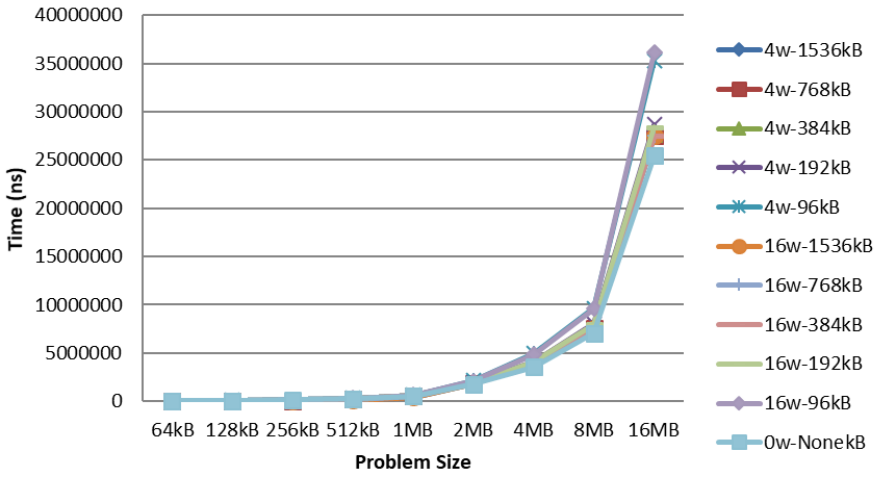


Figure 4.16: Graph of Simulation Time for Spatial Locality Benchmark. The size of L1 cache is 16 KB.

This graph shows that as the problem size gets larger, the time needed is also larger. This is to be expected, but the rate at which it grows is what is important. It is non-linear, meaning the rate that completion time increases is larger than the rate that the problem

size increases. This is not ideal for any computation. The same set of configurations were simulated again but with 32 KB of L1 cache to see if that had any impact on any of the metrics. While the performance was improved slightly with the increase of L1 cache, the same trends and overall poor performance were still present. This reiterates the results found previously that the size of L1 does not have such a large impact on performance.

Chapter 5

Conclusions and Future Work

5.1 Conclusion

This work mainly focused on the cache performance in a GPU when executing machine learning applications. By using a cycle accurate GPU simulator, detailed metrics about the memory architecture were obtained and analyzed. Rather than running full algorithms to get the complete set of metrics, suites of benchmarks that focused specifically on the GPU component of the algorithm were used. Two suites of benchmarks were used to judge the differences between the machine learning applications and the general purpose applications. Both sets of benchmarks consisted of programs that operated on matrices and on vectors.

A large component of the simulator was the ability to fully configure the GPU's memory architecture. By utilizing this feature of the simulator, modern GPU cache architectures could be emulated, and valid results could be obtained. The first set of data that came from the simulation of the benchmarks were with various L2 cache configurations. The goal of this was to determine if the machine learning benchmarks needed a large L2 cache by comparing the performance against the baseline of the general purpose benchmarks. The results of this were unexpected when the performance of a large subset of the benchmarks improved with the removal of L2 cache. Further tests and metrics were needed to explain this somewhat counterintuitive result.

To understand how the removal of L2 cache affects L1 performance, the cache coher-

ence protocol needed to be analyzed in great detail. Modifying the simulator to track the state transitions of the cache line led to the insight needed to explain the performance spike. In short, the small L2 cache had more misses which led to invalidation requests being sent up to the L1 caches. These requests then caused the data in L1 to be evicted and in turn caused more misses here too. Two conclusions were made about this discovery. The first is that the overhead with a small L2 cache hurts more than it helps so it is better to get rid of it all together. The second is that the NMOESI protocol isn't the best coherence protocol to implement in this case.

In addition to these tests, a couple more tests were run to further explain why the machine learning benchmarks had a larger performance spike than the general purpose benchmarks. Modifications to the benchmark problem sizes were made to ensure that the specific sizes weren't somehow affecting the results. All of the benchmarks from each suite were used to test a large range of input sizes. The results were very similar for both sets of benchmarks and showed the larger sizes tend to level off in performance.

The final test that was simulated was to verify the effect of spatial locality on the cache performance. This was investigated because machine learning algorithms have high spatial locality in their problems. By creating a new benchmark that had virtually no spatial locality by accessing random parts of the input, the effects were easily shown. The results were that the L1 hit ratio was virtually all misses. The L2 performance was good as long as the input size was smaller than the cache. Since this is not reasonable for full algorithms as they are much larger in size, the conclusion that higher spatial locality yields better cache performance was reinforced.

The overall conclusion of this work is many of the machine learning applications don't need a large L2 cache. Although a minority of the benchmarks had a large performance degradation, most of them had a very slight, if any, performance drop. This suggests the expensive and large footprint of L2 is not necessary when executing machine learning applications. The removal of L2 makes the manufacturing, cost, and running energy all

less while achieving the same performance.

5.2 Future Work

This work has some areas that could be expanded in future work. The largest area would be with the limitations of the GPU simulator. There were some workarounds that were used, but if a newer, fuller-featured simulator was available they wouldn't need to be used. In a similar fashion, larger benchmarks would provide more realistic data. The problem sizes used here are much smaller than actual problems that would be run on the GPU. If full algorithms could be run on the simulator, stronger conclusions could be made.

Expanding on this work, a new cache coherence protocol could be implemented to better fit the needs of the machine learning applications. One possibility would be to add a new state to the existing NMOESI protocol to better handle a small L2 cache if that is desired. Since a small L2 cache invalidates all of the copies in L1, this new state could be used to reduce the number of invalidations. Less invalidation requests sent to L1 would remove the valley that was shown to exist in the L1 cache performance. Another possibility is to use a completely different protocol that handles the cache behaviors exhibited by the machine learning benchmarks.

Finally, another expansion would be to change the cache architecture. This work discovered that the L2 may not be necessary to achieve the same performance in most machine learning applications. Another possible idea is to add a third layer to the cache architecture, to try to improve performance. This would make the second layer unify a subset of the L1 caches so an invalidation request issued by L2 doesn't evict all copies in every L1, rather just the set that it has access to. By limiting the number of L1 caches each L2 module is responsible for, the size of L2 could be reduced.

Bibliography

[1] D. R. Thambawita, R. Ragel, and D. Elkaduwe, "To use or not to use: Graphics processing units (GPUs) for pattern matching algorithms," in *2014 7th International Conference on Information and Automation for Sustainability: "Sharpening the Future with Sustainable Technology", ICIAfS 2014*, no. 7, pp. 1–5, 2014.

[2] A. K. Ziabari, Y. Sun, Y. Ma, D. Schaa, J. L. Abellán, R. Ubal, J. Kim, A. Joshi, and D. Kaeli, "UMH: A Hardware-Based Unified Memory Hierarchy for Systems with Multiple Discrete GPUs," *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 4, pp. 1–25, 2016.

[3] NVIDIA Corporation, "CUDA C Programming Guide: Design Guide," no. September, p. 247, 2015.

[4] Y. Nimkar, "Cache Memory Access Patterns in the GPU Architecture," 2018.

[5] H. Wen and W. Zhang, "Exploring GPU data cache leakage management techniques," *Journal of Computing Science and Engineering*, vol. 12, no. 3, pp. 106–114, 2018.

[6] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram, "Access Pattern-Aware Cache Management for Improving Data Utilization in GPU," *Proceedings of the 44th Annual International Symposium on Computer Architecture - ISCA '17*, pp. 307–319, 2017.

[7] "Many-core vs. many-thread machines: Stay away from the valley," *IEEE Computer Architecture Letters*, vol. 8, no. 1, pp. 25–28, 2009.

[8] NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210," *White Paper*, p. 23, 2014.

[9] NVIDIA Corporation, "NVIDIA Tesla V100 GPU Architecture," *White Paper*, no. v1.1, p. 53, 2017.

[10] X. Mei and X. Chu, "Dissecting GPU Memory Hierarchy Through Microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2017.

[11] NVIDIA Corporation, "NVIDIA Tesla P100 The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU," *White Paper*, p. 45, 2016.

[12] B. Lesage, D. Griffin, S. Altmeyer, L. Cucu-Grosjean, and R. I. Davis, "On the analysis of random replacement caches using static probabilistic timing methods for multi-path programs," *Real-Time Systems*, vol. 54, no. 2, pp. 307–388, 2018.

[13] A. Dan and D. Towsley, "An approximate analysis of the LRU and FIFO buffer replacement schemes," *ACM SIGMETRICS Performance Evaluation Review*, vol. 18, no. 1, pp. 143–152, 2007.

[14] P. G. De Massas and F. Pétrot, "Comparison of memory write policies for NoC based Multicore cache coherent Systems," *Proceedings -Design, Automation and Test in Europe, DATE*, no. March 2008, pp. 997–1002, 2008.

[15] G. Schindler, "Techniques for Caches in GPUs," pp. 1–7, 2016.

[16] M. H. Samavatian, H. Abbasitabar, M. Arjomand, and H. Sarbazi-Azad, "An Efficient STT-RAM Last Level Cache Architecture for GPUs," *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference - DAC '14*, pp. 1–6, 2014.

[17] F. Candel, S. Petit, J. Sahuquillo, and J. Duato, "Accurately modeling the on-chip and off-chip GPU memory subsystem," *Future Generation Computer Systems*, vol. 82, no. February, pp. 510–519, 2018.

[18] N. P. Jouppi, C. Young, N. Patil, D. Patterson, and G. Agrawal, "In-Datacenter Performance Analysis of a Tensor Processing Unit," *International Symposium on Computer Architecture*, pp. 1–12, 2017.

[19] Xilinx and Inc, "Accelerating DNNs with Xilinx Alveo Accelerator Cards (WP504)," vol. 504, pp. 1–11, 2018.

[20] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "DLAU: A scalable deep learning accelerator unit on FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 513–517, 2017.

[21] C. Barton, S. Chen, and Z. Chen, "The Multi2Sim Simulation Framework," 2015.

[22] NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," *White Paper*, p. 22.

[23] NVIDIA Corporation, "NVIDIA GeForce GTX 980 Featuring Maxwell, The Most Advanced GPU Ever Made," *White Paper*, p. 32, 2014.

[24] Multi2Sim, "M2s bench cudasdk 6.5." `https://github.com/Multi2Sim/m2s-bench-cudasdk-6.5`, 2016.

[25] D. Yu, H. Wang, P. Chen, and Z. Wei, "Mixed Pooling for Convolutional Neural Networks," *RSKT*, vol. 8818, pp. 364–375, 2014.

Appendix

NMOESI in M2S

Multi2Sim (M2S) implements the NMOESI protocol through a stack of system events. If an instruction is reading memory, an *event_load* is added to the stack. This event is handled by the `EventLoadHandler()`. Instructions that write to memory issue an *event_nc_store* to the stack. The handler for these events is `EventNCStoreHandler()`. Since M2S doesn't support atomic operations, there are no regular store events. The diagram of the two memory access handlers is shown in Figure 1. Each event is represented as a white box. Each event then advances to the next section of the handler by calling `Next()`, which adds a new event to the stack. Events can call a new internal event handler with `Call()`. This will start a new chain of events for the stack and upon completion, will return to the next section of the handler that called it. These are shown as colored hexagons in the diagram, and their corresponding handler diagrams are shown in Figures 2 and 3.

The two memory access event handlers start off very similar by terminating the event if the access can be coalesced with a previous memory access. If it cannot, the event advances to the next section where the event waits for older accesses. Once cleared, the event calls an internal event called *event_find_and_lock*. This handles finding the requested cache line and locking it so no other event can modify it. After this is where the two handlers start to differ. The load handler checks if the access was a hit or miss. On a hit, the block of data is retrieved and unlocked, and the event chain completes. On a miss, another call to an internal event handler is issued. This chain issues a read request to all caches for the block of data. When it is received, the load handler completes as if it was a hit. The non-coherent store is a little more complicated. After the cache line has been found and locked, it is evicted if it is in a *modified* or *owned* state since these states are responsible for writing data back once they are evicted. This process is handled by another internal event call. After that is done, there are three main paths for the event chain to take. The simplest is if the state is a hit in the *non-coherent* or *shared* states. The line can be written and unlocked,

Figure 1: Event Diagrams for Memory Access Handlers for the NMOESI Protocol as Implemented in Multi2Sim

and the event chain completed. The second path is if the state is *modified* or *owned*. In this case, a call to the *event_read_request* is issued. This is done to downgrade the states of other cache lines rather than receive data from them. After this the store can complete. The third and final path is similar to the last one but occurs when the state is *exclusive*. Instead of a read request, a call to remove the exclusive owner is done with the internal event *event_message*. Again, the store can complete after it returns.

The design of the internal event handlers are structured in a similar fashion to the memory access handlers. There are six internal event handlers used to implement the two access types. The first is the eviction handler. This is responsible for removing a valid cache line when a new line is going to replace it. This includes writing back data and invalidation of the cache line. The next internal event is the write request. This event handles the invalidation of other copies of a cache line when this cache line is going to modify the data. This is how the coherency is kept when modifying a cache line. The complement to this event is the read request. A read request is issued when a cache line needs to get data. The data can come either from another cache or from main memory. If it comes from another cache, that cache may need to change its state indicating that it is no longer the sole copy of that data. The chain of events to handle this type of request is the most complicated of all the event handlers. Both request types can be in either direction: from processor to memory or from memory to processor, which are known as UpDown and DownUp, respectively. The next event handler is shortest one. It is the invalidation event handler. All this handler does is issue write requests to upper level caches, if needed, and invalidate the specified cache line. The find and lock event handler is used in almost all handlers. As mentioned before, this finds the specified block and locks it so no other event can modify it until the event chain is complete. The final event handler is the message handler. This is a very linear event chain that is used only to remove the owners from an *exclusive* state, bringing it down to a *shared* state.

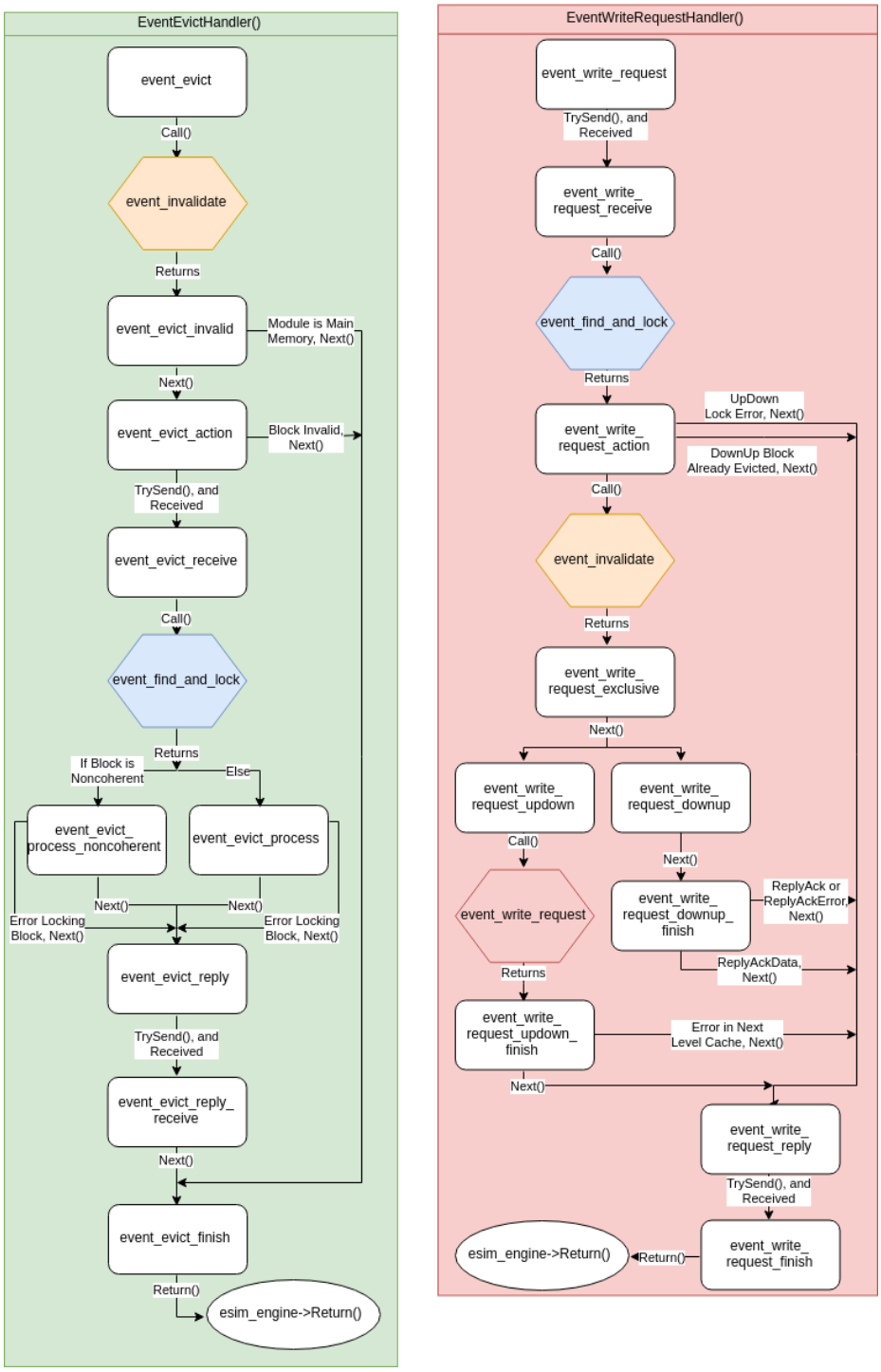


Figure 2: Event Diagrams for Internal Events for the NMOESI Protocol as Implemented in Multi2Sim

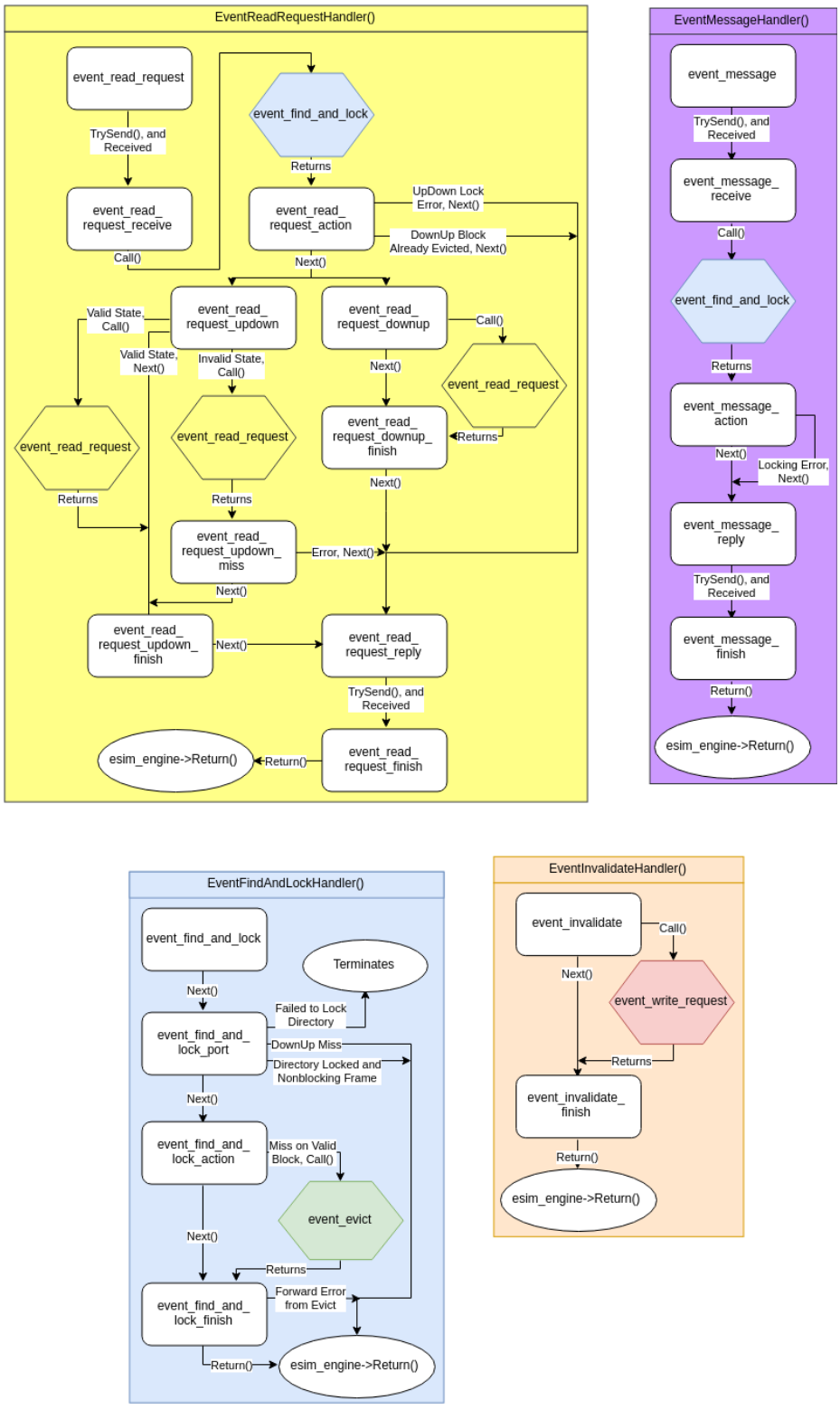


Figure 3: Event Diagrams for Internal Events for the NMOESI Protocol as Implemented in Multi2Sim